Jean-Pierre Deschamps · Elena Valderrama
Lluís Terés

Digital Systems

From Logic Gates to Processors

Springer

Digital Systems

Jean-Pierre Deschamps • Elena Valderrama
Lluís Terés

Digital Systems

From Logic Gates to Processors

Springer

Jean-Pierre Deschamps
School of Engineering
Rovira i Virgili University
Tarragona, Spain

Elena Valderrama
Escola d'Enginyeria
Campus de la UAB
Bellaterra, Spain

Lluís Terés
Microelectronics Institute of Barcelona
IMB-CNM (CSIC)
Campus UAB-Bellaterra, Cerdanyola
Barcelona, Spain

ISBN 978-3-319-41197-2 ISBN 978-3-319-41198-9 (eBook)
DOI 10.1007/978-3-319-41198-9

Library of Congress Control Number: 2016947365

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

Preface

Digital electronic components are present in almost all our private and professional activities:

- Our personal computers, our smartphones, or our tablets are made up of digital components such as microprocessors, memories, interface circuits, and so on.
- Digital components are also present within our cars, our TV sets, or even in our household appliances.
- They are essential components of practically any industrial production line.
- They are also essential components of public transport systems, of secure access control systems, and many others.

We could say that any activity involving:

- The acquisition of data from human interfaces or from different types of sensors
- The storage of data
- The transmission of data
- The processing of data
- The use of data to control human interfaces or to control different types of actuators (e.g., mechanical actuators), can be performed in a safe and fast way by means of Digital Systems.

Thus, nowadays digital systems constitute a basic technical discipline, essential to any engineer. That's the reason why the Engineering School of the Autonomous University of Barcelona (UAB) has designed an introductory course entitled "Digital Systems: From Logic Gates to Processors," available on the Coursera MOOC (Massive Open Online Course) platform. This book includes all the material presented in the above mentioned MOOC.

Digital systems are constituted of electronic circuits made up (mainly) of transistors. A transistor is a very small device, similar to a simple switch. On the other hand, a digital component, like a microprocessor, is a very large circuit able to execute very complex operations. How can we build such a complex system (a microprocessor) using very simple building blocks

(the transistors)? The answer to this question is the central topic of a complete course on digital systems.

This introductory course describes the basic methods used to develop digital systems, not only the traditional ones, based on the use of logic gates and flip-flops, but also more advanced techniques that permit to design very large circuits and are based on hardware description languages and simulation and synthesis tools.

At the end of this course the reader:

- Will have some idea of the way a new digital system can be developed, generally starting from a functional specification; in particular, she/he will be able to:
  - Design digital systems of medium complexity
  - Describe digital systems using a high-level hardware description language
  - Understand the operation of computers at their most basic level
- Will know the main problems the development engineer is faced with, during the process of developing a new circuit
- Will understand which design tools are necessary to develop a new circuit

This course addresses (at least) two categories of people: on the one hand, people interested to know what a digital system is and how it can be developed and nothing else, but also people who need some knowledge about digital systems as a previous step toward other technical disciplines, such as computer architecture, robotics, bionics, avionics, and others.

Overview

Chapter 1 gives a general definition of digital systems, presents generic description methods, and gives some information about the way digital systems can be implemented under the form of electronic circuits.

Chapter 2 is devoted to combinational circuits, a particular type of digital circuit (memoryless circuit). Among others, it includes an introduction to Boolean algebra, one of the mathematical tools used to define the behavior of digital circuits.

In Chap. 3, a particular type of circuit, namely, arithmetic circuits, is presented. Arithmetic circuits are present in almost any system so that they deserve some particular presentation. Furthermore, they constitute a first example of reusable blocks. Instead of developing systems from scratch, a common strategy in many technical disciplines is to reuse already developed parts. This modular approach is very common in software engineering and can also be considered in the case of digital circuits. As an example, think of building a multiplier using adders and one-digit multipliers.

Sequential circuits, which are circuits including memory elements, are the topic of Chap. 4. Basic sequential components (flip-flops) and basic building blocks (registers, counters, memories) are defined. Synthesis methods are

presented. In particular, the concept of finite state machines (FSM), a mathematical tool used to define the behavior of a sequential circuit, is introduced.

As an example of the application of the synthesis methods described all along in the previous chapters, the design of a complete digital system is presented in Chap. 5. It is a generic system, able to execute a set of algorithms, depending on the contents of a memory block that stores a program. This type of system is called a processor, in this case a very simple one.

The last two chapters are dedicated to more general considerations about design methods and tools (Chap. 6) and about physical implementations (Chap. 7).

All along the course, a standard hardware description language, namely, VHDL, is used to describe circuits. A short introduction to VHDL is included in Appendix A. In order to define algorithms, a more informal and not executable language (pseudocode) is used. It is defined in Appendix B. Appendix C is an introduction to the binary numeration system used to represent numbers.

Tarragona, Spain — Jean-Pierre Deschamps
Bellaterra, Spain — Elena Valderrama
Barcelona, Spain — Lluís Terés

Acknowledgments

The authors thank the people who have helped them in developing this book, especially Prof. Mercè Rullán who reviewed the text and is the author of Appendices B and C. They are grateful to the following institutions for providing them the means for carrying this work through to a successful conclusion: Autonomous University of Barcelona, National Center of Microelectronics (CSIC, Bellaterra, Spain), and University Rovira i Virgili (Tarragona, Spain).

Contents

About the Authors

Jean-Pierre Deschamps received an M.S. degree in electrical engineering from the University of Louvain, Belgium, in 1967; a Ph.D. in computer science from the Autonomous University of Barcelona, Spain, in 1983; and a Ph.D. degree in electrical engineering from the Polytechnic School of Lausanne, Switzerland, in 1984. He worked in several companies and universities. His research interests include ASIC and FPGA design and digital arithmetic. He is the author of ten books and more than a hundred international papers.

Elena Valderrama received an M.S. degree in physics from the Autonomous University of Barcelona (UAB), Spain, in 1975, and a Ph.D. in 1979. Later, in 2006, she got a degree in medicine from the same university. She is currently professor at the Microelectronics Department of the Engineering School of UAB. From 1980 to 1998, she was an assigned researcher in the IMB-CNM (CSIC), where she led several biomedical-related projects in which the design and integration of highly complex digital systems (VLSI) was crucial. Her current interests focus primarily on education, not only from the point of view of the professor but also in the management and quality control of engineering-related educational programs. Her research interests move around the biomedical applications of microelectronics.

Lluís Terés received an M.S. degree in 1982 and a Ph.D. in 1986, both in computer sciences, from the Autonomous University of Barcelona (UAB). He is working in UAB since 1982 and in IMB-CNM (CSIC) since its creation in 1985. He is head of the Integrated Circuits and Systems (ICAS) group at IMB with research activity in the fields of ASICs, sensor signal interfaces, body-implantable monitoring systems, integrated N/MEMS interfaces, flexible platform-based systems and SoC, and organic/printed microelectronics. He has participated in more than 60 industrial and research projects. He is coauthor of more than 70 papers and 8 patents. He has participated in two spin-offs. He is also a part time assistant professor at UAB.

Digital Systems 1

This first chapter divides up into three sections. The first section defines the concept of digital system. For that, the more general concept of physical system is first defined. Then, the particular characteristics of digital physical systems are presented. In the second section, several methods of digital system specification are considered. A correct and unambiguous initial system specification is a key aspect of the development work. Finally, the third section is a brief introduction to digital electronics.

1.1 Definition

As a first step, the more general concept of physical system is introduced. It is not easy to give a complete and rigorous definition of physical system. Nevertheless, this expression has a rather clear intuitive meaning, and some of their more important characteristics can be underlined.

A physical system could be defined as a set of interconnected objects or elements that realize some function and are characterized by a set of input signals, a set of output signals, and a relation between input and output signals. Furthermore, every signal is characterized by

- Its type, for example a voltage, a pressure, a temperature, and a switch state
- A range of values, for example all voltages between 0 and 1.5 V and all temperatures between 15 and 25 °C

Example 1.1 Consider the system of Fig. 1.1.
It controls the working of a boiler that is part of a room heating system and is connected to a mechanical selector that permits to define a reference temperature. A temperature sensor measures the ambient temperature. Thus, the system has two input signals

- *pos*: the selector position that defines the desired ambient temperature (any value between 10 and 30°)
- *temp*: the temperature measured by the sensor

and one output signal

- *onoff*, with two possible values *ON* (start the boiler) and *OFF* (stop the boiler).

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_1

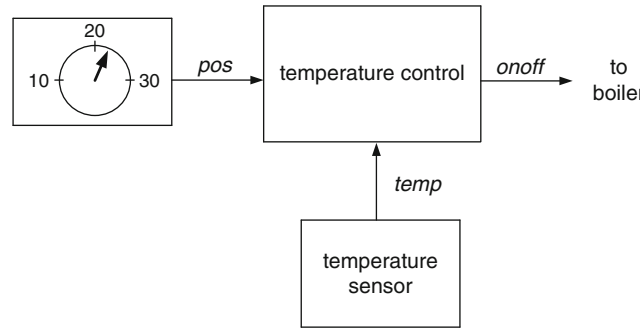


Fig. 1.1 Temperature control

The relation between inputs and output is defined by the following program in which *half_degree* is a previously defined constant equal to 0.5.

Algorithm 1.1 Temperature Control

```
loop
    if temp < pos - half_degree then onoff = on;
    elsif temp > pos + half_degree then onoff = off;
    end if;
    wait for 10 s;
end loop;
```

This is a pseudo-code program. An introduction to pseudo-code is given in Appendix B. However, this piece of program is quite easy to understand, even without any previous knowledge. Actually, the chosen pseudo-code is a simplified (non-executable) version of VHDL (Appendix A).

Algorithm 1.1 is a loop whose body is executed every 10 s: the measured temperature *temp* is compared with the desired temperature *pos* defined by the mechanical selector position; then

- If *temp* is smaller than *pos* − 0.5, then the boiler must get started so that the output signal *onoff* = *ON*.
- If *temp* is greater than *pos* + 0.5, then the boiler must be stopped so that the output signal *onoff* = *OFF*.
- If *temp* is included between *pos* − 0.5 and *pos* + 0.5, then no action is undertaken and the signal *onoff* value remains unchanged.

This is a functional specification including some additional characteristics of the final system. For example: The temperature updating is performed every 10 s, so that the arithmetic operations must be executed in less than 10 s, and the accuracy of the control is about ±0.5°.

As mentioned above, the type and range of the input and output signals must be defined.

- The input signal *temp* represents the ambient temperature measured by a sensor. Assume that the sensor is able to measure temperatures between 0 and 50°. Then *temp* is a signal whose type is "temperature" and whose range is "0 to 50°."
- The input signal *pos* is the position of a mechanical selector. Assume that it permits to choose any temperature between 10 and 30°. Then *pos* is a signal whose type is "position" and whose range is "10–30."
- The output signal *onoff* has only two possible values. Its type is "command" and its range is {*ON*, *OFF*}.

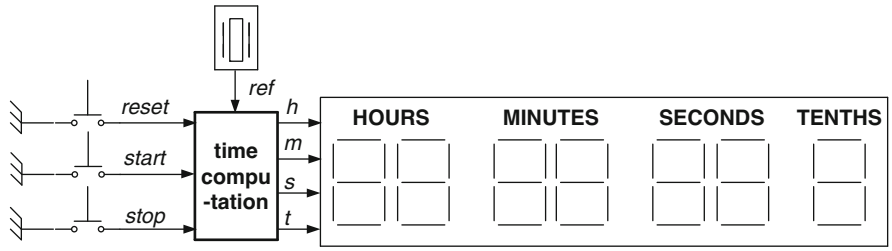


Fig. 1.2 Chronometer

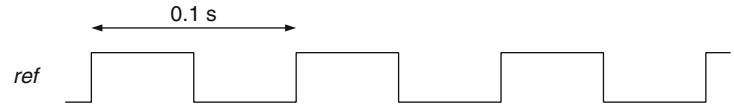


Fig. 1.3 Time reference signal

Assume now that the sensor is an ideal one, able to measure the temperature with an infinite accuracy, and that the selector is a continuous one, able to define the desired temperature with an infinite precision. Then both signals *temp* and *pos* are real numbers whose ranges are [0, 50] and [10, 30], respectively. Those signals, characterized by a continuous and infinite range of values, are called analog signals.

On the contrary, the range of the output signal *onoff* is a finite set $\{ON, OFF\}$. Signals whose range is a finite set (not necessary binary as in the case of *onoff*) are called digital signals or discrete signals.

Example 1.2 Figure 1.2 represents the structure of a chronometer.

- Three push buttons control its working. They generate binary (2-valued) signals *reset*, *start*, and *stop*.
- A crystal oscillator generates a time reference signal *ref* (Fig. 1.3): it is a square wave signal whose period is equal to 0.1 s (10 Hz).
- A *time computation* system computes the value of signals *h* (hours), *m* (minutes), *s* (seconds), and *t* (tenths of second).
- Some graphical interface displays the values of signals *h*, *m*, *s*, and *t*.

Consider the *time computation* block. It is a physical system (a subsystem of the complete chronometer) whose input signals are

- *reset*, *start*, and *stop* that are generated by three push buttons; according to the state of the corresponding switch, their value belongs to the set $\{closed, open\}$.
- *ref* is the signal generated by the crystal oscillator and is assumed to be an ideal square wave equal to either 0 or 1 V

and whose output signals are

- *h* belonging to the set $\{0, 1, 2, \ldots, 23\}$
- *m* and *s* belonging to the set $\{0, 1, 2, \ldots, 59\}$
- *t* belonging to the set $\{0, 1, 2, \ldots, 9\}$

The relation between inputs and outputs can be defined as follows (in natural language):

- When *reset* is pushed down then $h = m = s = t = 0$.
- When *start* is pushed down, the chronometer starts counting; h, m, s, and t represent the elapsed time in tenth of seconds.
- When *stop* is pushed down, the chronometer stops counting; h, m, s, and t represent the latest elapsed time.

In this example, all input and output signal values belong to finite sets. So, according to a previous definition, all input and output signals are digital. Systems whose all input and output signals are digital are called digital system.

1.2 Description Methods

In this section several specification methods are presented.

1.2.1 Functional Description

The relation between inputs and outputs of a digital system can be defined in a functional way, without any information about the internal structure of the system. Furthermore, a distinction can be made between explicit and implicit functional descriptions.

Example 1.3 Consider again the temperature controller of Example 1.1, with two modifications:

- The desired temperature (*pos*) is assumed to be constant and equal to 20° ($pos = 20$).
- The measured temperature has been discretized so that the signal *temp* values belong to the set $\{0, 1, 2, \ldots, 50\}$.

Then, the working of the controller can be described, in a completely explicit way, by Table 1.1 that associates to each value of *temp* the corresponding value of *onoff*: if *temp* is smaller than 20, then *onoff* = *ON*; if *temp* is greater than 20, then *onoff* = *ON*; if *temp* is equal to 20, then *onoff* keeps unchanged.

The same specification could be expressed by the following program.

Table 1.1 Explicit specification

temp	*onoff*
0	*ON*
1	*ON*
...	...
18	*ON*
19	*ON*
20	*unchanged*
21	*OFF*
22	*OFF*
...	...
49	*OFF*
50	*OFF*

Algorithm 1.2 Simplified Temperature Control

```
if temp < 20 then onoff = on;
elsif temp > 20 then onoff = off;
end if;
```

This type of description, by means of an algorithm, will be called "implicit functional description." In such a simple example, the difference between Table 1.1 and Algorithm 1.2 is only formal; in fact it is the same description. In more complex systems, a completely explicit description (a table) could be unmanageable.

Example 1.4 As a second example of functional specification consider a system (Fig. 1.4) that adds two 2-digit numbers.

Its input signals are

- x_1, x_0, y_1, and y_0 whose values belong to {0, 1, 2, . . ., 9}

and its output signals are

- z_2 whose values belong to {0, 1}, and z_1 and z_0 whose values belong to {0, 1, 2, . . ., 9}.

Digits x_1 and x_0 represent a number X belonging to the set {0, 1, 2, . . . , 99}; digits y_1 and y_0 represent a number Y belonging to the same set {0, 1, 2, . . . , 99}, and digits z_2, z_1, and z_0 represent a number Z belonging to the set {0, 1, 2, . . . ,198} where 198 = 99 + 99 is the maximum value of $X + Y$.

An explicit functional specification is Table 1.2 that contains 10,000 rows!

Another way to specify the function of a 2-digit adder is the following algorithm in which symbol/ stands for the integer division.

Fig. 1.4 2-Digit adder

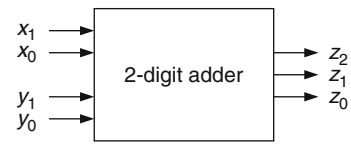


Table 1.2 Explicit specification of a 2-digit adder

$x_1\ x_0$	$y_1\ y_0$	$z_2\ z_1\ z_0$
00	00	000
00	01	001
. . .	. . .	. . .
00	99	099
01	00	001
01	01	002
. . .	. . .	. . .
01	99	100
. . .	. . .	. . .
99	00	099
99	01	100
. . .	. . .	. . .
99	99	198

Algorithm 1.3 2-Digit Adder

```
X = 10·x1 + x0;
Y = 10·y1 + y0;
Z = X + Y;
z2 = Z/100;
z1 = (Z - 100·z2)/10;
z0 = Z - 100·z2 - 10·z1;
```

As an example, if $x_1 = 5$, $x_0 = 7$, $y_1 = 7$, and $y_0 = 1$, then

$X = 10{\cdot}5 + 7 = 57.$
$Y = 10 \cdot 7 + 1 = 71.$
$Z = 57 + 71 = 128.$
$z_2 = 128/100 = 1.$
$z_2 = (128 - 100 \cdot 1)/10 = 28/10 = 2.$
$z_3 = 128 - 100 \cdot 1 - 10.2 = 8.$

At the end of the algorithm execution:

$$X + Y = Z = 100 \cdot z_2 + 10 \cdot z_1 + z_0.$$

Table 1.2 and Algorithm 1.3 are functional specifications. The first is explicit, the second is implicit, and both are directly deduced from the initial unformal definition: x_1 and x_0 represent X, digits y_1 and y_0 represent Y, and z_2, z_1, and z_0 represent $Z = X + Y$.

Another way to define the working of the 2-digit adder is to use the classical pencil and paper algorithm. Given two 2-digit numbers $x_1\ x_0$ and $y_1\ y_0$,

- Compute $s_0 = x_1 + x_0$.
- If $s_0 < 10$ then $z_0 = s_0$ and $carry = 0$; in the contrary case ($s_0 \geq 10$) then $z_0 = s_0 - 10$ and $carry = 1$.
- Compute $s_1 = y_1 + y_0 + carry$.
- If $s_1 < 10$ then $z_1 = s_1$ and $z_2 = 0$; in the contrary case ($s_1 \geq 10$) then $z_1 = s_1 - 10$ and $z_2 = 1$.

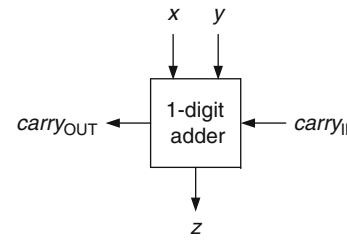
Algorithm 1.4 Pencil and Paper Algorithm

```
s0 = x0 + y0;
if s0 ≥ 10 then z0 = s0 - 10; carry = 1;
else z0 = s0; carry = 0;
end if;
s1 = x1 + y1 + carry;
if s1 ≥ 10 then z1 = s1 - 10; z2 = 1;
else z1 = s1; z2 = 0;
end if;
```

As an example, if $x_1 = 5$, $x_0 = 7$, $y_1 = 7$, and $y_0 = 1$, then

$s_0 = 7 + 1 = 8;$
$s_0 < 10$ so that $z_0 = 8$; $carry = 0$;

Fig. 1.5 1-Digit adder



$s_1 = 5 + 7 + 0 = 12$;
$s_1 \geq 10$ so that $z_1 = 12 - 10 = 2$; $z_2 = 1$;

and thus $57 + 71 = 128$.

Comment 1.1
Algorithm 1.4 is another implicit functional specification. However it is not directly deduced from the initial unformal definition as was the case of Table 1.2 and of Algorithm 1.3. It includes a particular step-by-step addition method and, to some extent, already gives some indication about the structure of the system (the subject of next Sect. 1.2.2). Furthermore, it could easily be generalized to the case of n-digit operands for any $n > 2$.

1.2.2 Structural Description

Another way to specify the relation between inputs and outputs of a digital system is to define its internal structure. For that, a set of previously defined and reusable subsystems called components must be available.

Example 1.5 Assume that a component called 1-digit adder (Fig. 1.5) has been previously defined. Its input signals are

- Digits x and y belonging to $\{0, 1, 2, \ldots, 9\}$
- $carry_{IN} \in \{0, 1\}$

and its output signals are

- $z \in \{0, 1, 2, \ldots, 9\}$
- $carry_{OUT} \in \{0, 1\}$

Every 1-digit adder component executes the operations that correspond to a particular step of the pencil and paper addition method (Algorithm 1.4):

- Add two digits and an incoming carry.
- If the obtained sum is greater than or equal to 10, subtract 10 and the outgoing carry is 1; in the contrary case the outgoing carry is 0.

The following algorithm specifies its working.

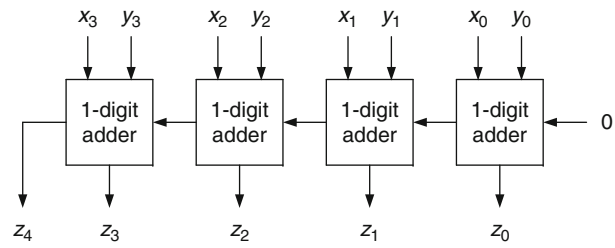


Fig. 1.6 4-Digit adder

Algorithm 1.5 1-Digit Adder

```
s = x + y + carry_IN;
if s ≥ 10 then z = s - 10; carry_OUT = 1;
else z = s; carry_OUT = 0;
end if;
```

With this component, the structure of a 4-digit adder can be defined (Fig. 1.6).

It computes the sum $Z = X + Y$ where $X = x_3\, x_2\, x_1\, x_0$ and $Y = y_3\, y_2\, y_1\, y_0$ are two 4-digit numbers and $Z = z_4\ \ z_3\ \ z_2\ \ z_1\ \ z_0$ is a 5-digit number whose most significant digit z_4 is 0 or 1 ($X + Y \leq 9999 + 9999 = 19{,}998$).

Comment 1.2
In the previous Example 1.5, four identical components (1-digit adders) are used to define a 4-digit adder by means of its structure (Fig. 1.6). The 1-digit adder in turn has been defined by its function (Algorithm 1.5). This is an example of 2-level hierarchical description. The first level is a diagram that describes the structure of the system, while the second level is the functional description of the components.

1.2.3 Hierarchical Description

Hierarchical descriptions with more than two levels can be considered. The following example describes a 3-level hierarchical description.

Example 1.6 Consider a system that computes the sum $z = w + x + y$ where w, x, and y are 4-digit numbers. The maximum value of z is 9999 + 9999 + 9999 = 29,997 that is a 5-digit number whose most significant digit is equal to 0, 1, or 2. The first hierarchical level (top level) is a block diagram with two different blocks (Fig. 1.7): a 4-digit adder and a 5-digit adder.

The 4-digit adder can be divided into four 1-digit adders (Fig. 1.8) and the 5-digit adder can be divided into five 1-digit adders (Fig. 1.9). Figures 1.8 and 1.9 constitute a second hierarchical level. Finally, a 1-digit adder (Fig. 1.5) can be defined by its functional description (Algorithm 1.5). It constitutes a third hierarchical level (bottom level).

Thus, the description of the system that computes z consists of three levels (Fig. 1.10). The lowest level is the functional description of a 1-digit adder. Assuming that 1-digit adder components are available, the system can be built with nine components.

A hierarchical description could be defined as follows.

- It is a set of interconnected blocks.
- Every block, in turn, is described either by its function or by a set of interconnected blocks, and so on.
- The final blocks correspond to available components defined by their function.

Fig. 1.7 Top level

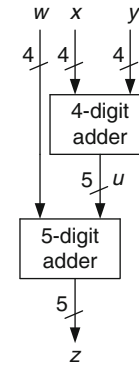


Fig. 1.8 4-Digit adder

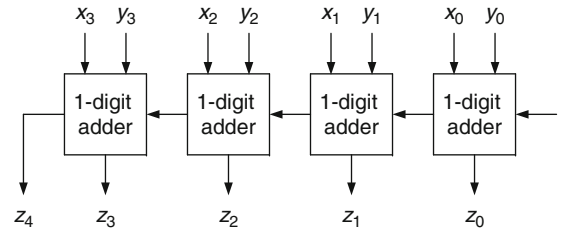
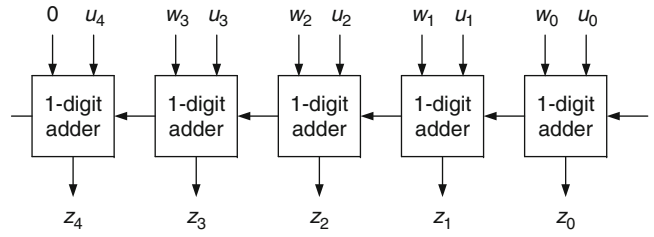


Fig. 1.9 5-Digit adder



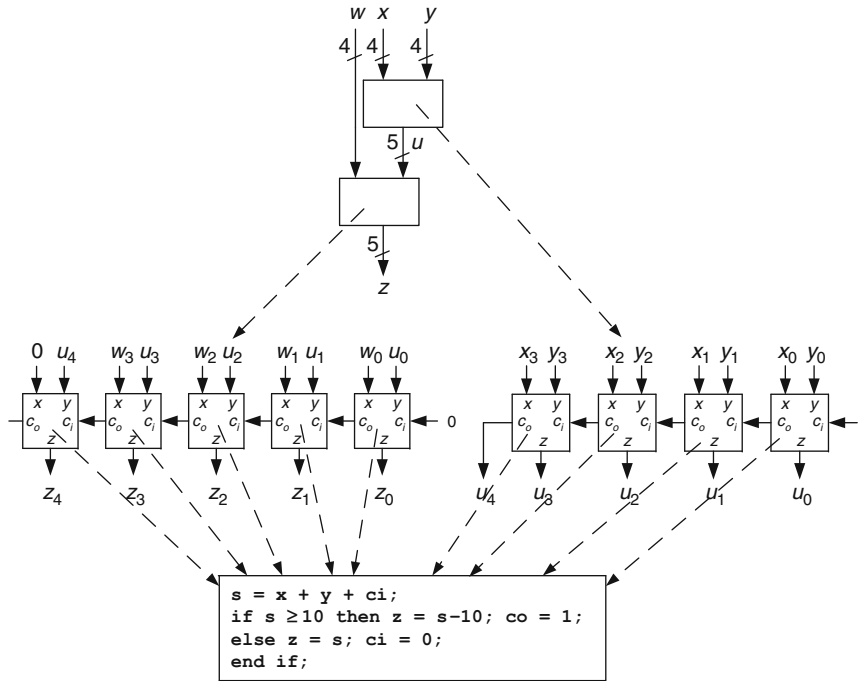


Fig. 1.10 Hierarchical description

Comments 1.3
Generally, the initial specification of a digital system is functional (a description of what the system does). In the case of very simple systems it could be a table that defines the output signal values in function of the input signal values. However, for more complex systems other specification methods should be used. A natural language description (e.g., in English) is a frequent option. Nevertheless, an algorithmic description (programing language, hardware description language, pseudo-code) could be a better choice: those languages have a more precise and unambiguous semantics than natural languages. Furthermore, programing language and hardware description language specifications can be compiled and executed, so that the initial specification can be tested. The use of algorithms to define the function of digital systems is one of the key aspects of this course.

In other cases, the initial specification already gives some information about the way the system must be implemented (see Examples 1.5 and 1.6).

In fact, the digital system designer work is the generation of a circuit made up of available components and whose behavior corresponds to the initial specification. Many times this work consists of successive refinements of an initial description: starting from an initial specification a (top level) block diagram is generated; then, every block is treated as a subsystem to which a more detailed block diagram is associated, and so on. The design work ends when all block diagrams are made up of interconnected components defined by their function and belonging to some available library of physical components (Chap. 7).

1.3 Digital Electronic Systems

The definition of digital system of Sect. 1.1 is a very general one and refers to any type of physical system whose input and output values belong to a finite set. In what follows, this course will focus on electronic systems.

1.3.1 Real System Structure

Most real digital systems include (Fig. 1.11)

- Input devices such as sensors, keyboards, microphones, and communication receivers.
- Output devices such as displays, motors, communication transmitters, and loudspeakers.

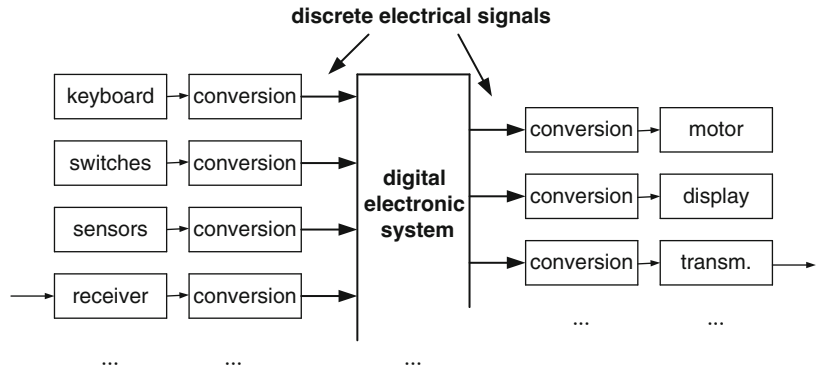


Fig. 1.11 Structure of a real digital system

- Input converters that translate the information generated by the input devices to discrete electrical signals.
- Output converters that translate discrete electrical signals into signals able to control the output devices.
- A digital electronic circuit—the brain of the system—that generates output electrical data in function of the input electrical data.

In Example 1.2, the input devices are three switches (push buttons) and a crystal oscillator, and the output device is a 7-digit display. The time computation block is an electronic circuit that constitutes the brain of the complete system.

Thus, real systems consist of a set of input and output interfaces that connect the input and output devices to the kernel of the system. The kernel of the system is a digital electronic system whose input and output signals are discrete electrical signals.

In most cases those input and output signals are binary encoded data. As an example, numbers can be encoded according to the binary numeration system and characters such as letters, digits, or some symbols can be encoded according to the standard ASCII codes (American Standard Code for Information Interchange).

1.3.2 Electronic Components

To build digital electronic systems, electronic components are used. In this section some basic information about digital electronic components is given. Much more complete and detailed information about digital electronics can be found in books such as Weste and Harris (2010) or Rabaey et al. (2003).

1.3.2.1 Binary Codification

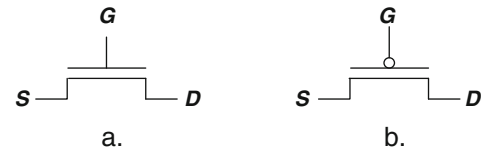
A first question: It has been mentioned above that, in most cases, the input and output signals are binary encoded data; but how are the binary digits (bits) 0 and 1 physically (electrically) represented? The usual solution consists in defining a low voltage V_L, and a high voltage V_H, and conventionally associating V_L to bit 0 and V_H to bit 1. The value of V_L and V_H depends on the implementation technology. In this section it is assumed that $V_L = 0$ V and $V_H = 1$ V.

1.3.2.2 MOS Transistors

Nowadays, most digital circuits are made up of interconnected MOS transistors. They are very small devices and large integrated circuits contain millions of transistors.

MOS transistors (Fig. 1.12a, b) have three terminals called S (source), D (drain), and G (gate). There are two types of transistors: n-type (Fig. 1.12a) and p-type (Fig. 1.12b) where n and p refer to the type of majority electrical charges (carriers) that can flow from terminal S (source) to terminal D (drain) under the control of the gate voltage: in an nMOS transistor the majority carriers are

Fig. 1.12 MOS transistors



electrons (negative charges) so that the current flows from D to S; in a pMOS transistor the majority carriers are holes (positive charges) so that the current flows from S to D.

A very simplified model (Fig. 1.13) is now used to describe the working of an nMOS transistor: it works like a switch controlled by the transistor gate voltage.

If the gate voltage V_G is low (0 V) then the switch is open (Fig. 1.14a, b) and no current could flow. If the gate voltage V_G is high (1 V) then the switch is closed (Fig. 1.14c, d) and V_{OUT} tends to be equal to V_{IN}. However, if V_{IN} is high (1 V) then V_{OUT} is not equal to 1 V (Fig. 1.14b). The maximum value of V_{OUT} is $V_G - V_T$ where the threshold voltage V_T is a characteristic of the implementation technology. It could be said that an nMOS transistor is a good switch for transmitting V_L (Fig. 1.14c), but not a good switch for transmitting V_H (Fig. 1.14d).

A similar model can be used to describe the working of a pMOS transistor. If the gate voltage V_G is high (1 V) then the switch is open (Fig. 1.15a, b) and no current could flow. If the gate voltage V_G is low (0 V) then the switch is closed (Fig. 1.15c, d) and V_{OUT} tends to be equal to V_{IN}. However, if V_{IN} is low (0 V) then V_{OUT} is not equal to 0 V (Fig. 1.15b). Actually the minimum value of V_{OUT} is $V_G + |V_T|$ where the threshold voltage V_T is a characteristic of the implementation technology. It could be said that a pMOS transistor is a good switch for transmitting V_H (Fig. 1.15c), but not a good switch for transmitting V_L (Fig. 1.15d).

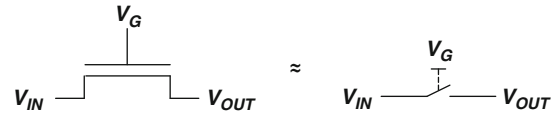


Fig. 1.13 Equivalent model

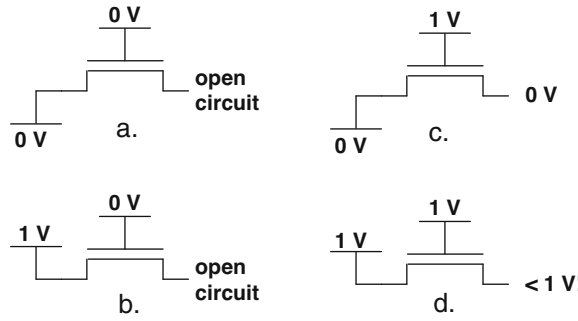


Fig. 1.14 nMOS switches

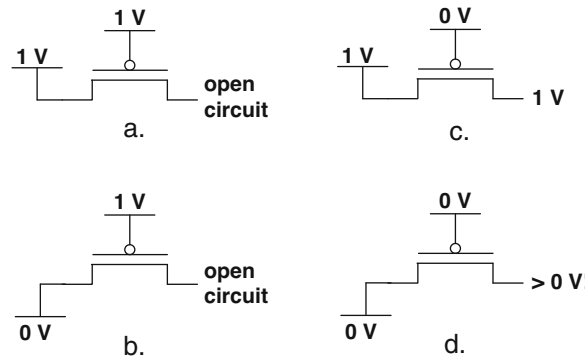


Fig. 1.15 pMOS switches

1.3.2.3 CMOS Inverter

By interconnecting several transistors, small components called logic gates can be implemented. The simplest one (Fig. 1.16) is the CMOS inverter, also called NOT gate.

A CMOS inverter consists of two transistors:

- A *p*MOS transistor whose source is connected to the high voltage V_H (1 V), whose gate is connected to the circuit input and whose drain is connected to the circuit output.
- An *n*MOS transistor whose source is connected to the low voltage V_L (0 V), whose gate is connected to the circuit input and whose drain is connected to the circuit output.

To analyze the working of this circuit in the case of binary signals, consider the two following input values:

- If $V_{IN} = 0$ V then (Fig. 1.17a) according to the simplified model of Sect. 1.3.2.2, the *n*MOS transistor is equivalent to an open switch and the *p*MOS transistor is equivalent to a closed switch (a good switch for transmitting V_H) so that $V_{OUT} = 1$ V.
- If $V_{IN} = 1$ V then (Fig. 1.17b) the *p*MOS transistor is equivalent to an open switch and the *n*MOS transistor is equivalent to a closed switch (a good switch for transmitting V_L) so that $V_{OUT} = 0$ V.

Fig. 1.16 CMOS inverter

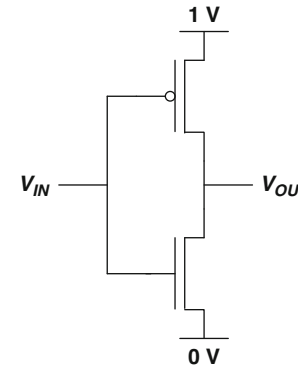
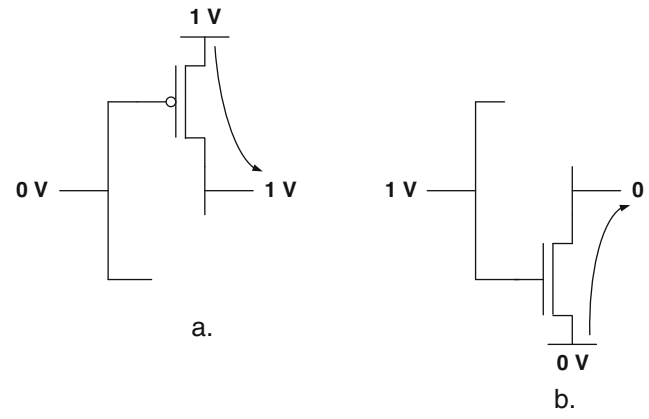


Fig. 1.17 Working of a CMOS inverter



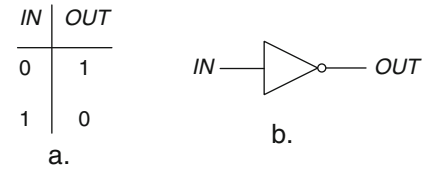


Fig. 1.18 Inverter: behavior and logic symbol

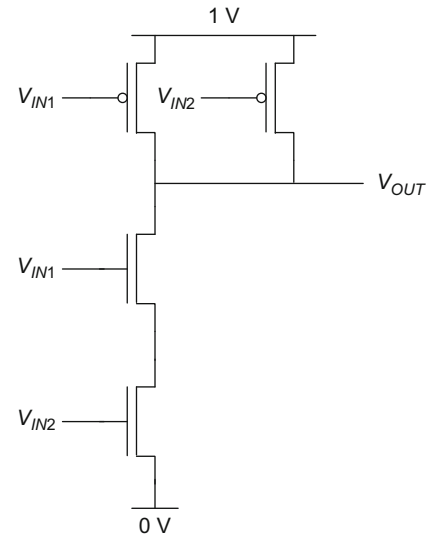


Fig. 1.19 2-Input NAND gate (NAND2 gate)

The conclusion of this analysis is that, as long as only binary signals are considered, the circuit of Fig. 1.16 inverts the input signal: it transforms V_L (0 V) into V_H (1 V) and V_H (1 V) into V_L (0 V). In terms of bits, it transforms 0 into 1 and 1 into 0 (Fig. 1.18a). As long as only the logic behavior is considered (the relation between input bits and output bits), the standard inverter symbol of Fig. 1.18b is used.

1.3.2.4 Other Components

With four transistors (Fig. 1.19) a 2-input circuit called NAND gate can be implemented. It works as follows:

- If $V_{IN1} = V_{IN2} = 1$ V then both *p*MOS switches are open and both *n*MOS switches are closed so that they transmit $V_L = 0$ V to the gate output (Fig. 1.20a).
- If $V_{IN2} = 0$ V, whatever the value of V_{IN1}, then at least one of the *n*MOS switches (connected in series) is open and at least one of the *p*MOS switches (connected in parallel) is closed, so that $V_H = 1$ V is transmitted to the gate output (Fig. 1.20b).
- If $V_{IN1} = 0$ V, whatever the value of V_{IN2}, the conclusion is the same.

Thus, the logic behavior of a 2-input NAND gate is given in Fig. 1.21a and the corresponding symbol is shown in Fig. 1.21b. The output of a 2-input NAND gate (NAND2) is equal to 0 if, and only if, both inputs are equal to 1. In all other cases the output is equal to 1.

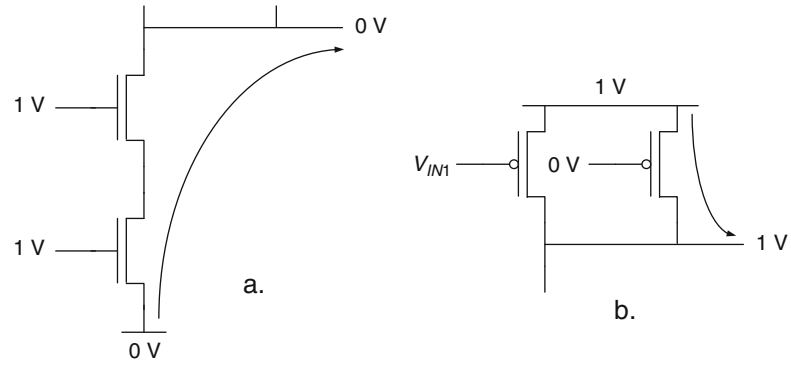


Fig. 1.20 NAND gate working

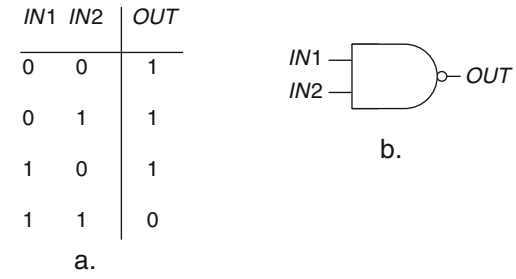


Fig. 1.21 2-Input NAND gate: behavior and symbol

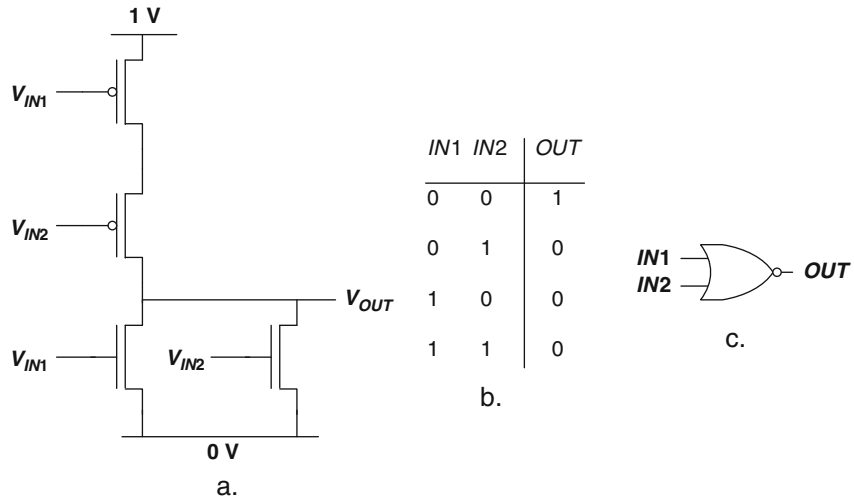


Fig. 1.22 NOR2 gate

Other logic gates can be defined and used as basic components of digital circuits. Some of them will now be mentioned. Much more complete information about logic gates can be found in classical books such as Floyd (2014) or Mano and Ciletti (2012).

The circuit of Fig. 1.22a is a 2-input NOR gate (NOR2 gate). If $V_{IN1} = V_{IN2} = 0$ V, then both p-type switches are closed and both n-type switches are open, so that $V_H = 1$ V is transmitted to the gate output. In all other cases at least one of the p-type switches is open and at least one of the n-type

switches is closed, so that $V_L = 0$ V is transmitted to the gate output. The logic behavior and the symbol of a NOR2 gate are shown in Fig. 1.22b, c.

NAND and NOR gates with more than two inputs can be defined. The output of a k-input NAND gate is equal to 0 if, and only if, the k inputs are equal to 1. The corresponding circuit (similar to Fig. 1.19) has k p-type transistors in parallel and k n-type transistors in series. The output of a k-input NOR gate is equal to 1 if, and only if, the k inputs are equal to 0. The corresponding circuit (similar to Fig. 1.22) has k n-type transistors in parallel and k p-type transistors in series. The symbol of a 3-input NAND gate (NAND3 gate) is shown in Fig. 1.23a and the symbol of a 3-input NOR gate (NOR3 gate) is shown in Fig. 1.23b.

The logic circuit of Fig. 1.24a consists of a NAND2 gate and an inverter. The output is equal to 1 if, and only if, both inputs are equal to 1 (Fig. 1.24b). It is a 2-input AND gate (AND2 gate) whose symbol is shown in Fig. 1.24c.

The logic circuit of Fig. 1.25a consists of a NOR2 gate and an inverter. The output is equal to 0 if, and only if, both inputs are equal to 0 (Fig. 1.25b). It is a 2-input OR gate (OR2 gate) whose symbol is shown in Fig. 1.25c.

AND and OR gates with more than two inputs can be defined. The output of a k-input AND gate is equal to 1 if, and only if, the k inputs are equal to 1, and the output of a k-input OR gate is equal to 0 if, and only if, the k inputs are equal to 0. For example, an AND3 gate can be implemented with a NAND3 gate and an inverter (Fig. 1.26a). Its symbol is shown in Fig. 1.26b. An OR3 gate can be implemented with a NOR3 gate and an inverter (Fig. 1.26c). Its symbol is shown in Fig. 1.26d.

Fig. 1.23 NAND3 and NOR3

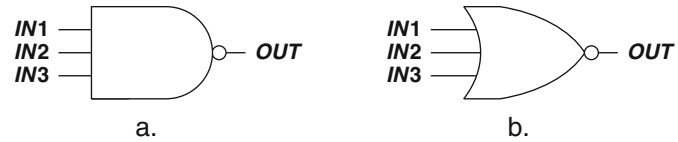


Fig. 1.24 AND2 gate

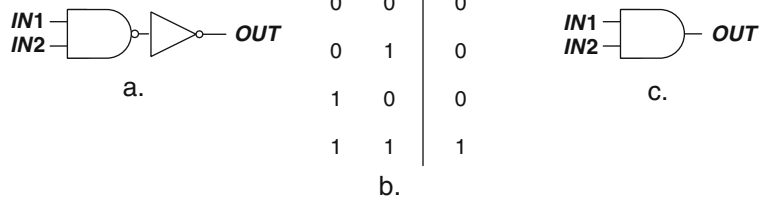
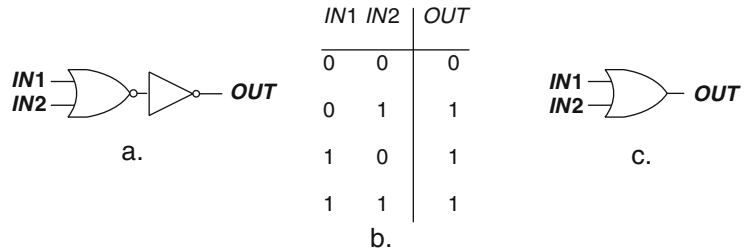


Fig. 1.25 OR2 gate



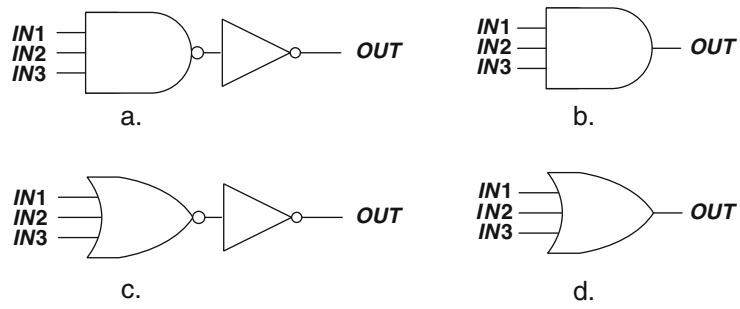


Fig. 1.26 AND3 and OR3 gates

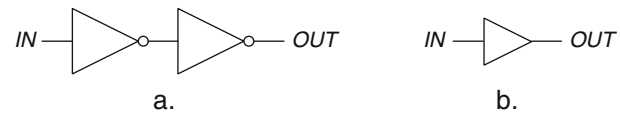


Fig. 1.27 Buffer

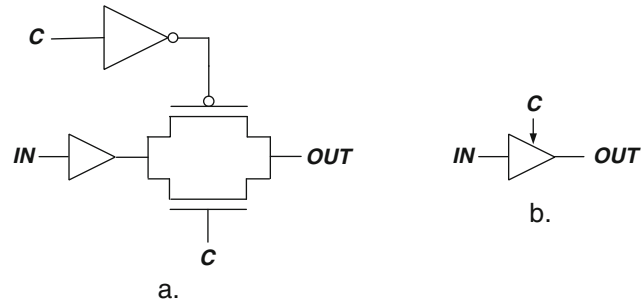


Fig. 1.28 3-State buffer

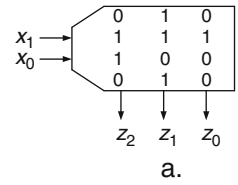
Buffers are another type of basic digital components. The circuit of Fig. 1.27a, made up of two inverters, generates an output signal equal to the input signal. Thus, it has no logic function; it is a power amplifier. Its symbol is shown in Fig. 1.27b.

The circuit of Fig. 1.28a is a 3-state buffer. It consists of a buffer, an inverter, a *p*MOS transistor, and an *n*MOS transistor. It has two inputs *IN* and *C* (control) and an output *OUT*. If $C = 0$, then both switches (*n*-type and *p*-type) are open, so that the output *OUT* is disconnected from the input *IN* (floating state or high impedance state). If $C = 1$, then both switches are closed, so that the output *OUT* is connected to the input *IN* through a good (*p*-type) switch if $IN = 1$ and through a good (*n*-type) switch if $IN = 0$. The 3-state buffer symbol is shown in Fig. 1.28b.

Other small-size components such as multiplexers, encoders, decoders, latches, flip flops, and others will be defined in the next chapters.

To conclude this section about digital components, an example of larger size component is given. Figure 1.29a is the symbol of a read-only memory (ROM) that stores four 3-bit words. Its behavior is specified in Fig. 1.29b: with two address bits x_1 and x_0 one of the four stored words is selected and can be read from outputs z_2, z_1, and z_0.

More generally, a ROM with N address bits and M output bits stores 2^N M-bit words (in total $M \cdot 2^N$ bits).



x_1	x_0	z_2	z_1	z_0
0	0	0	1	0
0	1	1	1	1
1	0	1	0	0
1	1	0	1	0

b.

Fig. 1.29 12-bit read-only memory ($3 \cdot 2^2$-bit ROM)

1.3.3 Synthesis of Digital Electronic Systems

The central topic of this course is the synthesis of digital electronic systems. The problem can be stated in the following way.

- On the one hand, the system designer has the specification of a system to be developed. Several specification methods have been proposed in Sect. 1.2.
- On the other hand, the system designer has a catalog of available electronic components such as logic gates, memories, and others, and might have access to previously developed and reusable subsystems. Some of the more common electronic components have been described in Sect. 1.3.2.

The designer work is the definition of a digital system that fulfils the initial specification and uses building blocks that belong to the catalog of available components or are previously designed subsystems. In a more formal way it could be said that the designer work is the generation of a hierarchical description whose final blocks are electronic components or reusable electronic subsystems.

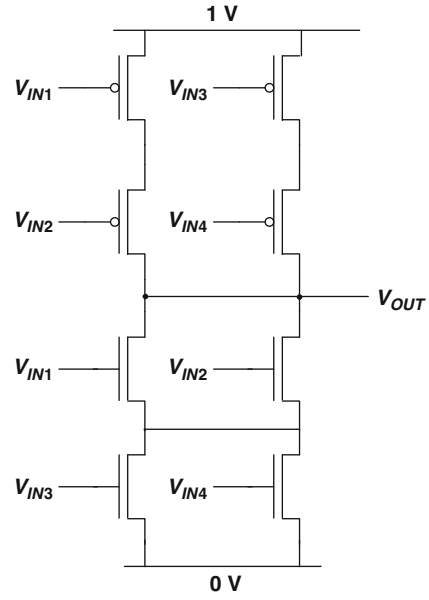
1.4 Exercises

1. The working of the chronometer of Example 1.2 can be specified by the following program in which the condition *ref_positive_edge* is assumed to be true on every positive edge of signal *ref*.

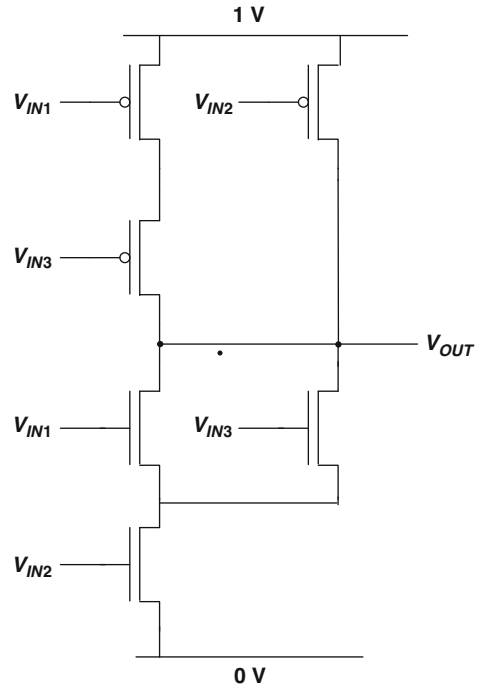
```
loop
  if reset = ON then h = 0; m = 0; s = 0; t = 0;
  elsif start = ON then
    while stop = OFF loop
      if ref_positive_edge = TRUE then
        update(h, m, s, t);
      end if;
    end loop;
  end if;
end loop;
```

The *update* procedure updates the values of h, m, s, and t every time that there is a positive edge on *ref*, that is to say every tenth of second. Generate a pseudo-code program that defines the *update* procedure.

2. Given two numbers X and $Y = y_3 \cdot 10^3 + y_2 \cdot 10^2 + y_1 \cdot 10 + y_0$, the product $P = X \cdot Y$ can be expressed as $P = y_0 \cdot X + y_1 \cdot X \cdot 10 + y_2 \cdot X \cdot 10^2 + y_3 \cdot X \cdot 10^3$. Generate a pseudo-code program based on the preceding relation to compute P.
3. Given two numbers X and $Y = y_3 \cdot 10^3 + y_2 \cdot 10^2 + y_1 \cdot 10 + y_0$, the product $P = X \cdot Y$ can be expressed as $P = (((y_3 \cdot X) \cdot 10 + y_2 \cdot X) \cdot 10 + y_1 \cdot X) \cdot 10 + y_0 \cdot X$. Generate a pseudo-code program based on the preceding relation to compute P.
4. Analyze the working of the following circuit and generate a 16-row table that defines V_{OUT} in function of V_{IN1}, V_{IN2}, V_{IN3}, and V_{IN4}.



5. Analyze the working of the following circuit and generate an 8-row table that defines V_{OUT} in function of V_{IN1}, V_{IN2}, and V_{IN3}.



References

Floyd TL (2014) Digital fundamentals. Prentice Hall, Upper Saddle River
Mano MMR, Ciletti MD (2012) Digital design. Prentice Hall, Boston
Rabaey JM, Chandrakasan A, Nikolic B (2003) Digital integrated circuits: a design perspective. Prentice Hall, Upper Saddle River
Weste NHE, Harris DM (2010) CMOS VLSI design: a circuit and systems perspective. Pearson, Boston

Combinational Circuits

2

Given a digital electronic circuit specification and a set of available components, how can the designer translate this initial specification to a circuit? The answer is the central topic of this course. In this chapter, an answer is given in the particular case of the combinational circuits.

2.1 Definitions

A switching function is a binary function of binary variables. In other words, an n-variable switching function associates a binary value, 0 or 1, to any n-component binary vector. As an example, in Fig. 1.29, z_2, z_1, and z_0 are three 2-variable switching functions.

A digital circuit that implements a set of switching functions in such a way that at any time the output signal values only depend on the input signal values at the same moment is called a combinational circuit. The important point of this definition is "at the same moment." A combinational circuit with n inputs and m outputs is shown in Fig. 2.1. It implements m switching functions

$$f_i\colon \{0,\ 1\}^n \rightarrow \{0,\ 1\},\ i = 0,\ 1,\ \ldots, m-1.$$

To understand the condition "at the same moment" an example of circuit that is not combinational is now given.

Example 1.2 (A Non-combinational Circuit) Consider the temperature controller of Example 1.3 defined by Table 1.1 and substitute *ON* by 1 and *OFF* by 0. The obtained Table 2.1 does not define a combinational circuit: the knowledge that the current temperature is 20° does not permit to decide whether the output signal must be 0 or 1. To decide, it is necessary to know the previous value of the temperature. In other words, this circuit must have some kind of memory.

Example 2.2 Consider the 4-bit adder of Fig. 2.2. Input bits x_3, x_2, x_1, and x_0 represent an integer X in binary numeration (Appendix C); input bits y_3, y_2, y_1, and y_0 represent another integer Y, input bit c_i is an incoming carry; and output bits z_4, z_3, z_2, z_1, and z_0 represent an integer Z. The relation between inputs and outputs is

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_2

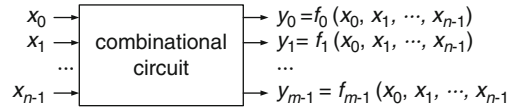
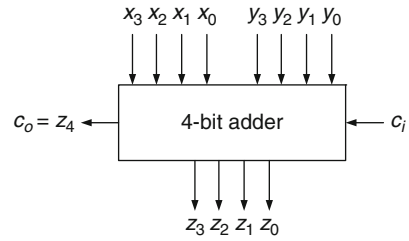


Fig. 2.1 *n*-Input *m*-output combinational circuit

Table 2.1 Specification of a non-combinational circuit

Temp	Onoff
0	1
1	1
...	...
18	1
19	1
20	*Unchanged*
21	0
22	0
...	...
49	0
50	0

Fig. 2.2 4-Bit adder



$$Z = X + Y + c_i.$$

Observe that X and Y are 4-bit integers included within the range of 0–15, so that the maximum value of Z is $15 + 15 + 1 = 31$ that is a 5-bit number. Output z_4 could also be used as an outgoing carry c_o.

In this example, the value of the output bits only depends on the value of the input bits at the same time; it is a combinational circuit.

2.2 Synthesis from a Table

A completely explicit specification of a 4-bit adder (Fig. 2.2) is a table that defines five switching functions z_4, z_3, z_2, z_1, and z_0 of nine variables x_3, x_2, x_1, x_0, y_3, y_2, y_1, y_0, and c_i (Table 2.2).

A straightforward implementation method consists in storing the Table 2.2 contents in a read-only memory (Fig. 2.3). The address bits are the input signals x_3, x_2, x_1, x_0, y_3, y_2, y_1, y_0, and c_i and the stored words define the value of the output signals z_4, z_3, z_2, z_1, and z_0. As an example, if the address bits are 100111001, so that $x_3x_2x_1x_0 = 1001$, $y_3y_2y_1y_0 = 1100$, and $c_i = 1$, then $X = 9$, $Y = 12$, and $Z = 9 + 12 + 1 = 22$, and the stored word is 10110 that is the binary representation of 22.

Obviously this is a universal synthesis method: it can be used to implement any combinational circuit. The generic circuit of Fig. 2.1 can be implemented by the ROM of Fig. 2.4. However, in many

Table 2.2 Explicit specification of a 4-bit adder

$x_3x_2x_1x_0$	$y_3y_2y_1y_0$	c_i	$z_4z_3z_2z_1z_0$
0000	0000	0	00000
0000	0000	1	00001
0000	0001	0	00001
0000	0001	1	00010
0000	0010	0	00010
0000	0010	1	00011
...	...	...	...
1001	1100	1	10110
...	...	...	...
1111	1111	0	11110
1111	1111	1	11111

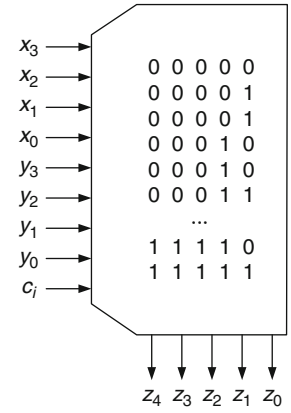


Fig. 2.3 ROM implementation of a 4-bit adder

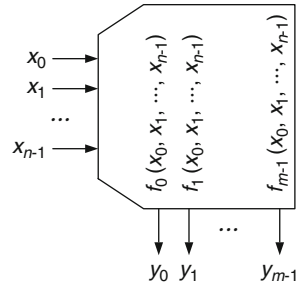


Fig. 2.4 ROM implementation of a combinational circuit

cases this is a very inefficient implementation method: the ROM of Fig. 2.4 must store $m \cdot 2^n$ bits, generally a (too) big number. As an example the ROM of Fig. 2.3 stores $5 \cdot 2^9 = 2{,}560$ bits.

Instead of using a universal, but inefficient, synthesis method, a better option is to take advantage of the peculiarities of the system under development. In the case of the preceding Example 2.2 (Fig. 2.2), a first step is to divide the 4-bit adder into four 1-bit adders (Fig. 2.5).

Each 1-bit adder is a combinational circuit that implements two switching functions z and d of three variables x, y, and c. Each 1-bit adder executes the operations that correspond to a particular step of the binary addition method (Appendix C). A completely explicit specification is given in Table 2.3.

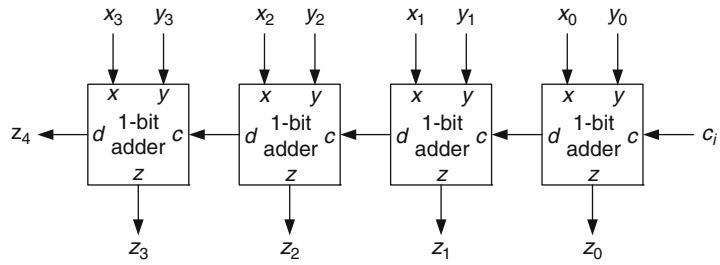


Fig. 2.5 Structure of a 4-bit adder

Table 2.3 Explicit specification of a 1-bit adder

x	y	c	d	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

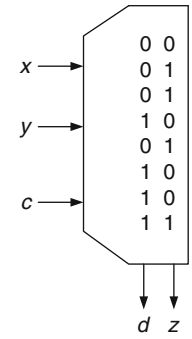


Fig. 2.6 ROM implementation of a 1-bit adder

In this case, a ROM implementation could be considered (Fig. 2.6). This type of small ROM (eight 2-bit words in this example) is often called lookup table (LUT) and it is the method used in field programmable gate arrays (FPGA) to implement switching functions of a few variables (Chap. 7).

Instead of a ROM, a table can also be implemented by means of logic gates (Sect. 1.3.2), for example AND gates, OR gates, and Inverters (or NOT gates). Remember that

- The output of an n-input AND gate is equal to 1 if, and only if, its n inputs are equal to 1.
- The output of an n-input OR gate is equal to 1 if, and only if, at least one of its n inputs is equal to 1.
- The output of an inverter is equal to 1 if, and only if, its input is equal to 0.

Define now a 3-input switching function $p(x, y, c)$ as follows: $p = 1$ if, and only if, $x = 1$, $y = 0$, and $c = 1$ (Table 2.4).

Table 2.4 Explicit specification of p

x	y	c	p
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

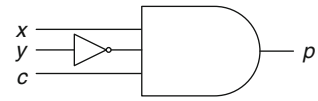


Fig. 2.7 Implementation of p

Table 2.5 Explicit specification of p_1, p_2, p_3, and p_4

x	y	c	p_1	p_2	p_3	p_4
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

This function p is implemented by the circuit of Fig. 2.7: the output of the AND3 gate is equal to 1 if, and only if, $x = 1$, $c = 1$ and the inverter output is equal to 1, that is, if $y = 0$.

The function d of Table 2.3 can be defined as follows: d is equal to 1 if, and only if, one of the following conditions is true:

$$x = 0, y = 1, c = 1,$$
$$x = 1, y = 0, c = 1,$$
$$x = 1, y = 1, c = 0,$$
$$x = 1, y = 1, c = 1.$$

The switching functions p_1, p_2, p_3, and p_4 can be associated to those conditions (Table 2.5). Actually, the function p of Table 2.4 is the function p_2 of Table 2.5.

Each function p_i can be implemented in the same way as p (Fig. 2.7) as shown in Fig. 2.8.

Finally, the function d can be defined as follows: d is equal to 1 if, and only if, one of the functions p_i is equal to 1. The corresponding circuit is a simple OR4 gate (Fig. 2.9) and the complete circuit is shown in Fig. 2.10.

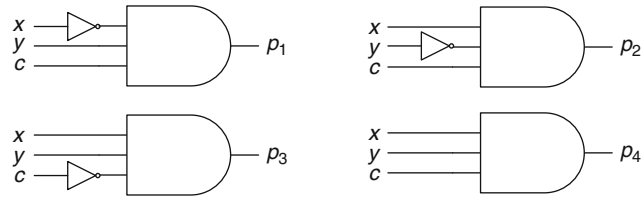


Fig. 2.8 Implementation of p_1, p_2, p_3, and p_4

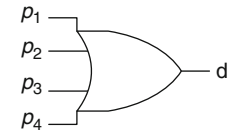


Fig. 2.9 Implementation of d

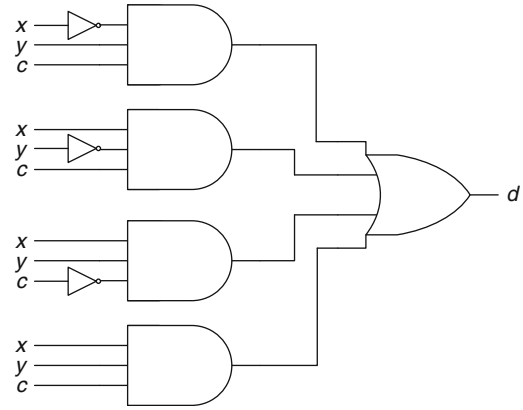


Fig. 2.10 Complete circuit

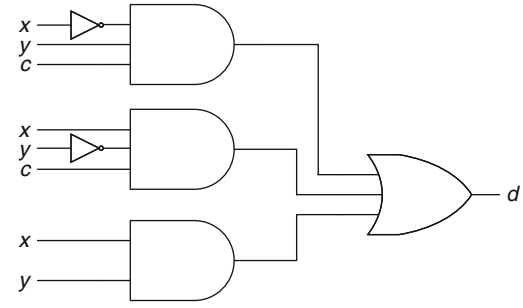


Fig. 2.11 Simplified circuit

Comment 2.1

The conditions implemented by functions p_3 and p_4 are $x = 1, y = 1, c = 0$ and $x = 1, y = 1, c = 1$, and can obviously be substituted by the simple condition $x = 1$ and $y = 1$, whatever the value of c. Thus, in Fig. 2.10 two of the four AND3 gates can be replaced by a single AND2 gate (Fig. 2.11).

The synthesis with logic gates of z (Table 2.3) is left as an exercise.

In conclusion, given a combinational circuit whose initial specification is a table, two possible options are:

- To store the table contents in a ROM
- To translate the table to a circuit made up of logic gates

Furthermore, in the second case, some optimization must be considered: if the inverters are not taken into account, the circuit of Fig. 2.11 contains 4 logic gates and 11 logic gate inputs, while the circuit of Fig. 2.10 contains 5 logic gates and 16 logic gate inputs. In CMOS technology (Sect. 1.3.2) the number of transistors is equal to twice the number of gate inputs so that the latter could be used as a measure of the circuit complexity.

A conclusion is that a tool that helps to minimize the number of gates and the number of gate inputs is necessary. It is the topic of the next section.

2.3 Boolean Algebra

Boolean algebra is a mathematical support used to specify and to implement switching functions. Only finite Boolean algebras are considered in this course.

2.3.1 Definition

A Boolean algebra B is a finite set over which two binary operations are defined:

- The Boolean sum $+$
- The Boolean product

Those operations must satisfy six rules (postulates).

The Boolean sum and the Boolean product are internal operations:

$$\forall a \text{ and } b \in B : a + b \in B \text{ and } a \cdot b \in B. \tag{2.1}$$

Actually, this postulate only emphasizes the fact that $+$ and $\cdot$ are operations over B.

The set B includes two particular (and different) elements 0 and 1 that satisfy the following conditions:

$$\forall a \in B : a + 0 = a \text{ and } a \cdot 1 = a. \tag{2.2}$$

In other words, 0 and 1 are neutral elements with respect to the sum (0) and with respect to the product (1).

Every element of B has an inverse in B:

$$\forall\, a \in B,\ \exists\, \overline{a} \in B \text{ such that } a + \overline{a} = 1 \text{ and } a \cdot \overline{a} = 0. \tag{2.3}$$

Both operations are commutative:

$$\forall a \text{ and } b \in B : a + b = b + a \text{ and } a \cdot b = b \cdot a. \tag{2.4}$$

Both operations are associative:

$$\forall a, b \text{ and } c \in B, a \cdot (b \cdot c) = (a \cdot b) \cdot c \text{ and } a + (b + c) = (a + b) + c. \tag{2.5}$$

The product is distributive over the sum and the sum is distributive over the product:

$$\forall a, b \text{ and } c \in B, a \cdot (b + c) = a \cdot b + a \cdot c \text{ and } a + b \cdot c = (a + b) \cdot (a + c). \tag{2.6}$$

Comment 2.2
Rules (2.1)–(2.6) constitute a set of symmetric postulates: given a rule, by interchanging sum and product, and 0 and 1, another rule is obtained: for example the fact that $a + 0 = a$ implies that $a \cdot 1 = a$, or the fact that $a \cdot (b + c) = a \cdot b + a \cdot c$ implies that $a + b \cdot c = (a + b) \cdot (a + c)$. This property is called duality principle.

The simplest example of Boolean algebra is the set $B_2 = \{0, 1\}$ with the following operations (Table 2.6):

- $a + b = 1$ if, and only if, $a = 1$ or $b = 1$, the OR function of a and b
- $a + b = 1$ if, and only if, $a = 1$ and $b = 1$, the AND function of a and b
- The inverse of a is $1 - a$

It can easily be checked that all postulates are satisfied. As an example (Table 2.7) check that the product is distributive over the sum, that is, $a \cdot (b + c) = a \cdot b + a \cdot c$.

The relation between B_2 and the logic gates defined in Sect. 1.3.2 is obvious: an AND gate implements a Boolean product, an OR gate implements a Boolean sum, and an inverter (NOT gate) implements an invert function (Fig. 2.12).

Table 2.6 Operations over B_2

a	b	$a + b$	$a \cdot b$	$\bar{a}$
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	0

Table 2.7 $a \cdot (b + c) = a \cdot b + a \cdot c$

a	b	c	$b + c$	$a \cdot (b + c)$	$a \cdot b$	$a \cdot c$	$a \cdot b + b \cdot c$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

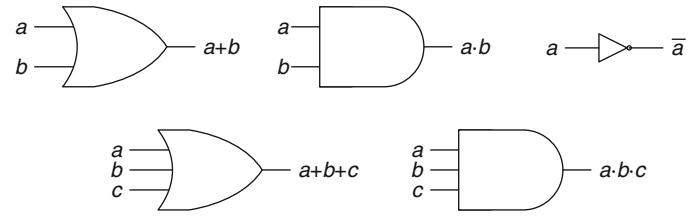


Fig. 2.12 Logic gates and Boolean functions

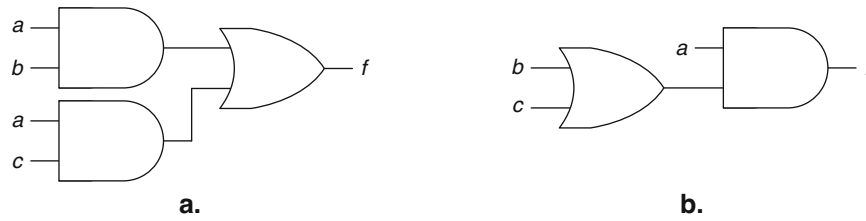


Fig. 2.13 Two equivalent circuits

In fact, there is a direct relation between Boolean expressions and circuits. As an example, a consequence of the distributive property $a \cdot b + a \cdot c = a \cdot (b + c)$ is that the circuits of Fig. 2.13 implement the same switching function, say f. However, the circuit that corresponds to the Boolean expression $a \cdot b + a \cdot c$ (Fig. 2.13a) has three gates and six gate inputs while the other (Fig. 2.13b) includes only two gates and three gate inputs. This is a first (and simple) example of how Boolean algebra helps the designer to optimize circuits.

Other finite Boolean algebras can be defined. Consider the set $B_2{}^n = \{0, 1\}^n$, that is, the set of all $2^n n$-component binary vectors. It is a Boolean algebra in which product, sum, and inversion are component-wise operations:

$$\begin{aligned} \forall a = (a_0, a_1, \ldots, a_{n-1}) \text{ and } b = (b_0, b_1, \ldots, b_{n-1}) \in B_2{}^n : \\ a + b = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1}), \\ a \cdot b = (a_0 \cdot b_0, a_1 \cdot b_1, \ldots, a_{n-1} \cdot b_{n-1}), \\ \overline{a} = (\overline{a_0}, \overline{a_1}, \ldots, \overline{a_{n-1}}). \end{aligned}$$

The neutral elements are $0 = (0, 0, \ldots, 0)$ and $1 = (1, 1, \ldots, 1)$.

Another example is the set of all subsets of a finite set S. Given two subsets S_1 and S_2, their sum is $S_1 \cup S_2$ (union), their product is $S_1 \cap S_2$ (intersection), and the inverse of S_1 is $S \backslash S_1$ (complement of S_1 with respect to S). The neutral elements are the empty set Ø and S. If S has n elements the number of subsets of S is 2^n.

A third example, the most important within the context of this course, is the set of all n-variable switching functions. Given two switching functions f and g, functions $f + g, f \cdot g$, and $\overline{f}$ are defined as follows:

$$\begin{aligned} &\forall (x_0, x_1, \ldots, x_{n-1}) \in B_2{}^n : \\ &(f + g)(x_0, x_1, \ldots, x_{n-1}) = f(x_0, x_1, \ldots, x_{n-1}) + g(x_0, x_1, \ldots, x_{n-1}), \\ &(f \cdot g)(x_0, x_1, \ldots, x_{n-1}) = f(x_0, x_1, \ldots, x_{n-1}) \cdot g(x_0, x_1, \ldots, x_{n-1}), \\ &\overline{f}(x_0, x_1, \ldots, x_{n-1}) = \overline{f(x_0, x_1, \ldots, x_{n-1})}. \end{aligned}$$

The neutral elements are the constant functions 0 and 1.

Comment 2.3

Mathematicians have demonstrated that any finite Boolean algebra is isomorphic to $B_2{}^m$ for some $m > 0$. In particular, the number of elements of any finite Boolean algebra is a power of 2. Consider the previous examples.

1. The set of subsets of a finite set $S = \{s_1, s_2, \ldots, s_n\}$ is a Boolean algebra isomorphic to $B_2{}^n$: associate to every subset S_1 of S an n-component binary vector whose component i is equal to 1 if, and only if, $s_i \in S_1$, and check that the vectors that correspond to the union of two subsets, the

intersection of two subsets, and the complement of a subset are obtained by executing the component-wise addition, the component-wise product, and the component-wise inversion of the associated n-component binary vectors.

2. The set of all n-variable switching functions is a Boolean algebra isomorphic to ${B_2}^m$ with $m = 2^n$: associate a number i to each of the 2^n elements of $\{0, 1\}^n$ (for example the natural number represented in binary numeration by this vector); then, associate to any n-variable switching function f a 2^n-component vector whose component number i is the value of f at point i. In the case of functions d and z of Table 2.3, $n = 3$, $2^n = 8$, and the 8-component vectors that define d and z are (00010111) and (01101001), respectively.

2.3.2 Some Additional Properties

Apart from rules (2.1–2.6) several additional properties can be demonstrated and can be used to minimize Boolean expressions and to optimize the corresponding circuits.

Properties 2.1

1. $\overline{0} = 1$ and $\overline{1} = 0$. (2.7)

2. Idempotence : $\forall a \in B : a + a = a$ and $a \cdot a = a$. (2.8)

3. $\forall a \in B : a + 1 = 1$ and $a \cdot 0 = 0$. (2.9)

4. Inverse uniqueness : if $a \cdot b = 0, a + b = 1, a \cdot c = 0$ and $a + c = 1$ then $b = c$. (2.10)

5. Involution : $\forall a \in B : \overline{\overline{a}} = a$. (2.11)

6. Absorption law : $\forall a$ and $b \in B, a + a \cdot b = a$ and $a \cdot (a + b) = a$. (2.12)

7. $\forall a$ and $b \in B$, $a + \overline{a} \cdot b = a + b$ and $a \cdot (\overline{a} + b) = a \cdot b$. (2.13)

8. de Morgan laws : $\forall a$ and $b \in B, \overline{a + b} = \overline{a} \cdot \overline{b}$ and $\overline{a \cdot b} = \overline{a} + \overline{b}$. (2.14)

9. Generalized de Morgan laws : $\forall a_1, a_2, \ldots, a_n \in B$, (2.15)

$$\overline{a_1 + a_2 + \ \ldots \ + a_n} = \overline{a_1} \cdot \overline{a_2} \cdot \ \ldots \ \cdot \overline{a_n} \text{ and } \overline{a_1 \cdot a_2 \cdot \ \ldots \ \cdot a_n} = \overline{a_1} + \overline{a_2} + \ \ldots \ + \overline{a_n}.$$

Proof

1. $\overline{0} = 0 + \overline{0} = 1$ and $\overline{1} = 1 \cdot \overline{1} = 0$.

2. $a = a + 0 = a + (a \cdot \overline{a}) = (a + a) \cdot (a + \overline{a}) = (a + a) \cdot 1 = a + a$; $a = a \cdot 1 = a \cdot (a + \overline{a})$ $= (a \cdot a) + (a \cdot \overline{a}) = (a \cdot a) + 0 = a \cdot a$.

3. $a + 1 = a + a + \overline{a} = a + \overline{a} = 1; a \cdot 0 = a \cdot a \cdot \overline{a} = a \cdot \overline{a} = 0.$

4. $b = b \cdot (a + c) = a \cdot b + b \cdot c = 0 + b \cdot c = a \cdot c + b \cdot c = (a + b) \cdot c = 1 \cdot c = c.$

5. Direct consequence of (4).

6. $a + a \cdot b = a \cdot 1 + a \cdot b = a \cdot (1 + b) = a \cdot 1 = a; a \cdot (a + b) = a \cdot a + a \cdot b = a + a \cdot b = a.$

7. $a + \overline{a} \cdot b = (a + \overline{a}) \cdot (a + b) = 1 \cdot (a + b) = a + b; a \cdot (\overline{a} + b) = (a \cdot \overline{a}) + (a \cdot b) = 0 + (a \cdot b) = a \cdot b.$

8. $(a + b) \cdot \overline{a} \cdot \overline{b} = a \cdot \overline{a} \cdot \overline{b} + b \cdot \overline{a} \cdot \overline{b} = 0 \cdot \overline{b} + 0 \cdot \overline{a} = 0 + 0 = 0;$
$(a + b) + \overline{a} \cdot \overline{b} = a + \left(b + \overline{a} \cdot \overline{b}\right) = a + b + \overline{a} = b + 1 = 1.$

9. By induction.

2.3.3 Boolean Functions and Truth Tables

Tables such as Table 2.3 that defines two switching functions d and z are called truth tables. If f is an n-variable switching function then its truth table has 2^n rows, that is, the number of different n-component vectors.

In this section the relation between Boolean expressions, truth tables, and gate implementation of combinational circuits is analyzed.

Given a Boolean expression, that is a well-constructed expression using variables and Boolean operations (sum, product, and inversion), a truth table can be defined. For that the value of the expression must be computed for every combination of variable values, in total 2^n different combinations if there are n variables.

Example 2.3 Consider the following Boolean expression that defines a 3-variable switching function f:

$$f(a, b, c) = b \cdot \overline{c} + \overline{a} \cdot b.$$

Define a table with as many rows as the number of combinations of values of a, b, and c, that is, $2^3 = 8$ rows, and compute the value of f that corresponds to each of them (Table 2.8).

Table 2.8 $f(a, b, c) = b \cdot \overline{c} + \overline{a} \cdot b.$

abc	$\overline{c}$	$b \cdot \overline{c}$	$\bar{a}$	$\overline{a} \cdot b$	$f = b \cdot \overline{c} + \overline{a} \cdot b$
000	1	0	1	0	0
001	0	0	1	0	0
010	1	1	1	1	1
011	0	0	1	1	1
100	1	0	0	0	0
101	0	0	0	0	0
110	1	1	0	0	1
111	0	0	0	0	0

Conversely, a Boolean expression can be associated to any truth table. For that, first define some new concepts.

Definitions 2.1

1. A literal is a variable or the inverse of a variable. For example a, $\overline{a}$, b, $\overline{b}$, ... are literals.
2. An n-variable minterm is a product of n literals such that each variable appears only once.

For example, if $n = 3$ then there are eight different minterms:

$$\begin{aligned} &m_0 = \overline{a} \cdot \overline{b} \cdot \overline{c}, m_1 = \overline{a} \cdot \overline{b} \cdot c, m_2 = \overline{a} \cdot b \cdot \overline{c},\ m_3 = \overline{a} \cdot b \cdot c, \\ &m_4 = a \cdot \overline{b} \cdot \overline{c}, m_5 = a \cdot \overline{b} \cdot c, m_6 = a \cdot b \cdot \overline{c}, m_7 = a \cdot b \cdot c. \end{aligned} \tag{2.16}$$

Their corresponding truth tables are shown in Table 2.9. Their main property is that to each minterm m_i is associated one, and only one, combination of values of a, b, and c such that $m_i = 1$:

$$\begin{aligned} &m_0 \text{ is equal to 1 if. and only if}, abc = 000, \\ &m_1 \text{ is equal to 1 if. and only if}, abc = 001, \\ &m_2 \text{ is equal to 1 if. and only if}, abc = 010, \\ &m_3 \text{ is equal to 1 if. and only if}, abc = 011, \\ &m_4 \text{ is equal to 1 if. and only if}, abc = 100, \\ &m_5 \text{ is equal to 1 if. and only if}, abc = 101, \\ &m_6 \text{ is equal to 1 if. and only if}, abc = 110, \\ &m_7 \text{ is equal to 1 if. and only if}, abc = 111. \end{aligned}$$

In other words, $m_i = 1$ if, and only if, abc is equal to the binary representation of i.

Consider now a 3-variable function f defined by its truth table (Table 2.10). From Table 2.9 it can be deduced that $f = m_2 + m_3 + m_6$, and thus (2.16)

$$f = \overline{a} \cdot b \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c}. \tag{2.17}$$

More generally, the n-variable minterm $m_i(x_{n-1}, x_{n-2}, \ldots, x_0)$ is equal to 1 if, and only if, the value of $x_{n-1}x_{n-2} \ldots x_0$ is the binary representation of i. Given a truth table that defines an n-variable switching function $f(x_{n-1}, x_{n-2}, \ldots, x_0)$, this function is the sum of all minterms m_i such that $i_{n-1}i_{n-2} \ldots i_0$ is the binary representation of i and $f(i_{n-1}, i_{n-2}, \ldots, i_0) = 1$. This type of representation of a switching function under the form of a Boolean sum of minterms (like (2.17)) is called canonical representation.

Table 2.9 3-Variable minterms

abc	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
000	1	0	0	0	0	0	0	0
001	0	1	0	0	0	0	0	0
010	0	0	1	0	0	0	0	0
011	0	0	0	1	0	0	0	0
100	0	0	0	0	1	0	0	0
101	0	0	0	0	0	1	0	0
110	0	0	0	0	0	0	1	0
111	0	0	0	0	0	0	0	1

Table 2.10 Truth table of f

abc	f
000	0
001	0
010	1
011	1
100	0
101	0
110	1
111	0

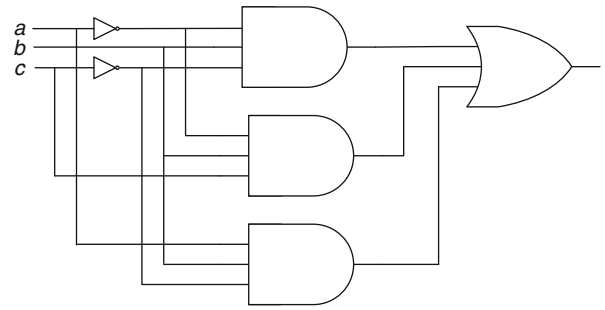


Fig. 2.14 $f = \overline{a} \cdot b \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c}$

The relation between truth table and Boolean expression, namely canonical representation, has been established. From a Boolean expression, for example (2.17), a circuit made up of logical gates can be deduced (Fig. 2.14).

Another example: the functions p_1, p_2, p_3, and p_4 of Table 2.5 are minterms of variables x, y, and c, and the circuit of Fig. 2.10 corresponds to the canonical representation of d.

Assume that a combinational system has been specified by some functional description, for example an algorithm (an implicit functional description). The following steps permit to generate a logic circuit that implements the function.

- Translate the algorithm to a table (an explicit functional description); for that, execute the algorithm for all combinations of the input variable values.
- Generate the canonical representation that corresponds to the table.
- Optimize the expression using properties of the Boolean algebras.
- Generate the corresponding circuit made up of logic gates.

As an example, consider the following algorithm that defines a 3-variable switching function.

Algorithm 2.1 Specification of $f(a, b, c)$

```
if (a=1 and b=1 and c=0) or (a=0 and b=1) then f = 1;
else f = 0;
end if;
```

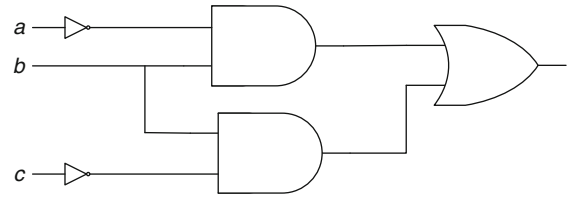


Fig. 2.15 Optimized circuit

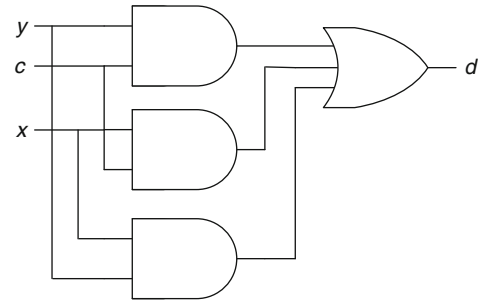


Fig. 2.16 Implementation of (2.20)

By executing this algorithm for each of the eight combinations of values of a, b, and c, Table 2.10 is obtained. The corresponding canonical expression is (2.17). This expression can be simplified using Boolean algebra properties:

$$\overline{a} \cdot b \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c} = \overline{a} \cdot b \cdot (\overline{c} + c) + (\overline{a} + a) \cdot b \cdot \overline{c} = \overline{a} \cdot b + b \cdot \overline{c}.$$

The corresponding circuit is shown in Fig. 2.15. It implements the same function as the circuit of Fig. 2.14, with fewer gates and fewer gate inputs. This is an example of the kind of circuit optimization that Boolean algebras permit to execute.

2.3.4 Example

The 4-bit adder of Sect. 2.2 is now revisited and completed. A first step is to divide the 4-bit adder into four 1-bit adders (Fig. 2.5). Each 1-bit adder implements two switching functions d and z defined by their truth tables (Table 2.3). The canonical expressions that correspond to the truth tables of d and z are the following:

$$d = \overline{x} \cdot y \cdot c + x \cdot \overline{y} \cdot c + x \cdot y \cdot \overline{c} + x \cdot y \cdot c, \tag{2.18}$$

$$z = \overline{x} \cdot \overline{y} \cdot c + \overline{x} \cdot y \cdot \overline{c} + x \cdot \overline{y} \cdot \overline{c} + x \cdot y \cdot c. \tag{2.19}$$

The next step is to optimize the Boolean expressions. Equation 2.18 can be optimized as follows:

$$d = (\overline{x} + x) \cdot y \cdot c + x \cdot (y + \overline{y}) \cdot c + x \cdot y \cdot (c + \overline{c}) = y \cdot c + x \cdot c + x \cdot y. \tag{2.20}$$

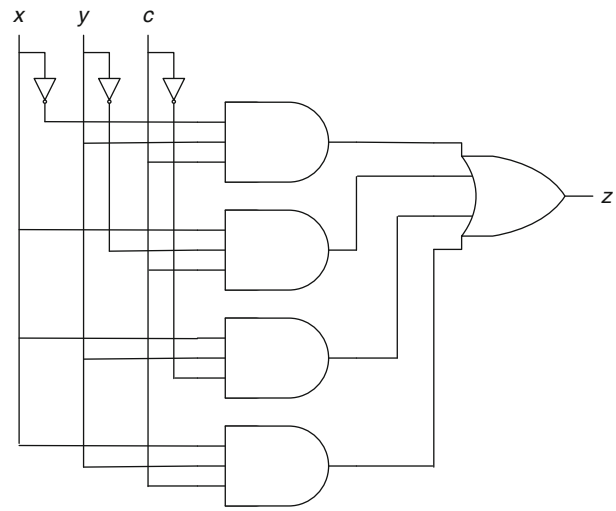


Fig. 2.17 Implementation of (2.19)

The corresponding circuit is shown in Fig. 2.16. It implements the same function d as the circuit of Fig. 2.11, with fewer gates, fewer gate inputs, and without inverters.

Equation 2.19 cannot be simplified. The corresponding circuit is shown in Fig. 2.17.

2.4 Logic Gates

In Sects. 2.2 and 2.3 a first approach to the implementation of switching functions has been proposed. It is based on the translation of the initial specification to Boolean expressions. Then, circuits made up of AND gates, OR gates, and inverters can easily be defined. However, there exist other components (Sect. 1.3.2) that can be considered to implement switching functions.

2.4.1 NAND and NOR

NAND gates and NOR gates have been defined in Sect. 1.3.2. They can be considered as simple extensions of the CMOS inverter and are relatively easy to implement in CMOS technology.

A NAND gate is equivalent to an AND gate and an inverter, and a NOR gate is equivalent to an OR gate and an inverter (Fig. 2.18).

The truth tables of a 2-input NAND function and of a 2-input NOR function are shown in Figs. 1.21a and 1.22b, respectively. More generally, the output of a k-input NAND gate is equal to 0 if, and only if, the k inputs are equal to 1, and the output of a k-input NOR gate is equal to 1 if, and only if, the k inputs are equal to 0. Thus,

$$\mathrm{NAND}(x_1, x_2, \ldots, x_n) = \overline{x_1 \cdot x_2 \cdot \ \ldots \cdot x_n} = \overline{x_1} + \overline{x_2} + \ldots + \overline{x_n}\,, \tag{2.21}$$

$$\mathrm{NOR}(x_1, x_2, \ldots, x_n) = \overline{x_1 + x_2 + \ldots + x_n} = \overline{x_1} \cdot \overline{x_2} \cdot \ldots \cdot \overline{x_n}\,. \tag{2.22}$$

Sometimes, the following algebraic symbols are used:

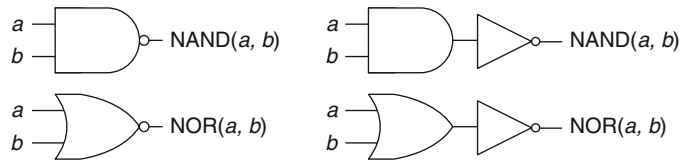


Fig. 2.18 NAND2 and NOR2 symbols and equivalent circuits

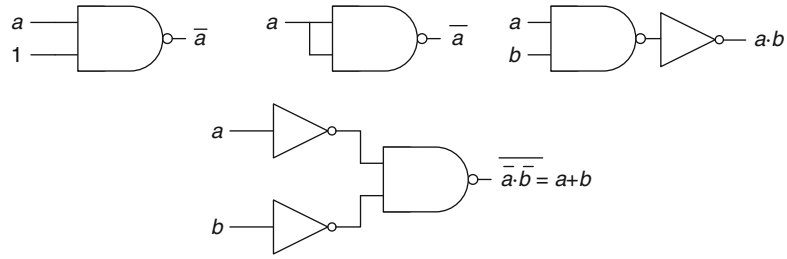


Fig. 2.19 NOT, AND2, and OR2 gates implemented with NAND2 gates and inverters

$$a \uparrow b = \text{NAND}(a, b) \text{ and } a \downarrow b = \text{NOR}(a, b).$$

NAND and NOR gates are universal modules. That means that any switching function can be implemented only with NAND gates or only with NOR gates. It has been seen in Sect. 2.3 that any switching function can be implemented with AND gates, OR gate, and inverters (NOT gates). To demonstrate that NAND gates are universal modules, it is sufficient to observe that the AND function, the OR function, and the inversion can be implemented with NAND functions. According to (2.21)

$$x_1 \cdot x_2 \cdot \ldots \cdot x_n = \overline{\overline{x_1 \cdot x_2 \cdot \ldots \cdot x_n}} = \overline{\text{NAND}(x_1, x_2, \ldots, x_n)}, \tag{2.23}$$

$$x_1 + x_2 + \ldots + x_n = \text{NAND}(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}), \tag{2.24}$$

$$\overline{x} = \overline{x \cdot 1} = \text{NAND}(x, 1) = \overline{x \cdot x} = \text{NAND}(x, x). \tag{2.25}$$

As an example NOT, AND2, and NOR2 gates implemented with NAND2 gates are shown in Fig. 2.19.

Similarly, to demonstrate that NOR gates are universal modules, it is sufficient to observe that the AND function, the OR function, and the inversion can be implemented with NOR functions. According to (2.22)

$$x_1 + x_2 + \ldots + x_n = \overline{\overline{x_1 + x_2 + \ldots + x_n}} = \overline{\text{NOR}(x_1, x_2, \ldots, x_n)}, \tag{2.26}$$

$$x_1 \cdot x_2 \cdot \ldots \cdot x_n = \text{NOR}(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}), \tag{2.27}$$

$$\overline{x} = \overline{x + 0} = \text{NOR}(x, 0) = \overline{x + x} = \text{NOR}(x, x). \tag{2.28}$$

Example 2.4 Consider the circuit of Fig. 2.11. According to (2.23) and (2.24), the AND gates and the OR gate can be substituted by NAND gates. The result is shown in Fig. 2.20a. Furthermore, two serially connected inverters can be substituted by a simple connection (Fig. 2.20b).

Comments 2.4

1. Neither the 2-variable NAND function (NAND2) nor the 2-variable NOR function (NOR2) are associative operations. For example

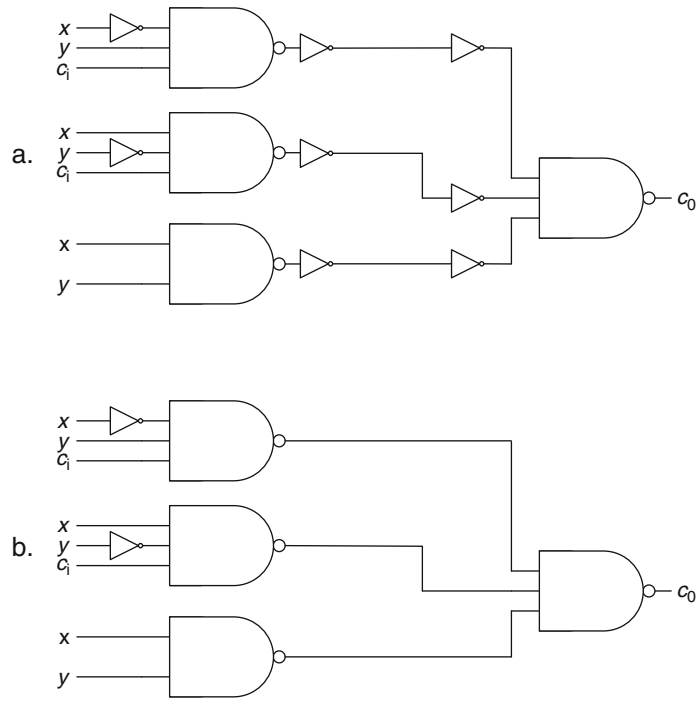


Fig. 2.20 Circuits equivalent to Fig. 2.11

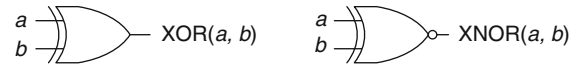


Fig. 2.21 XOR gate and XNOR gate symbols

$$\text{NAND}(a, \text{NAND}(b, c)) = \overline{a} + \overline{\text{NAND}(b, c)} = \overline{a} + b \cdot c,$$
$$\text{NAND}(\text{NAND}(a, b), c) = a \cdot b + \overline{c},$$

and none of the previous functions is equal to $NAND(a, b, c) = \overline{a} + \overline{b} + \overline{c}$.

2. As already mentioned above, NAND gates and NOR gates are easy to implement in CMOS technology. On the contrary, AND gates and OR gates must be implemented by connecting a NAND gate and an inverter or a NOR gate and an inverter, respectively. Thus, within a CMOS integrated circuit, NAND gates and NOR gates use less silicon area than AND gates and OR gates.

2.4.2 XOR and XNOR

XOR gates, where XOR stands for eXclusive OR, and XNOR gates are other commonly used components, especially in arithmetic circuits.

The 2-variable XOR switching function is defined as follows:

Table 2.11 XOR and XNOR truth tables

$a\ b$	XOR(a, b)	XNOR(a, b)
00	0	1
01	1	0
10	1	0
11	0	1

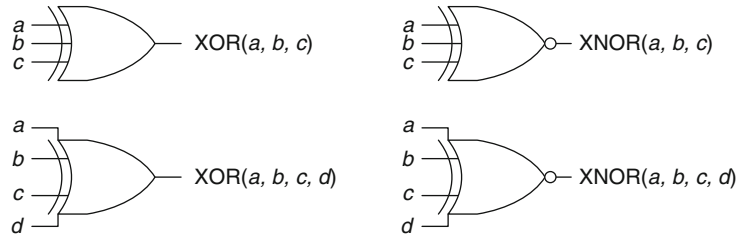


Fig. 2.22 3-Input and 4-input XOR gates and XNOR gates

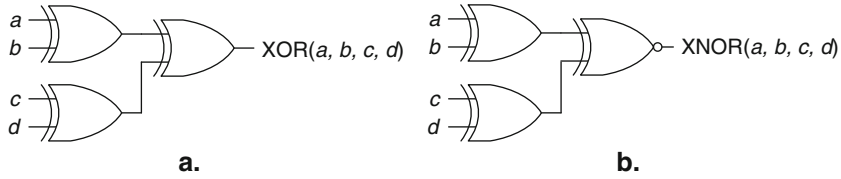


Fig. 2.23 4-Input XOR and XNOR gates implemented with 2-input gates

$$\text{XOR}(a,b) = 1 \text{ if, and only if,} a \neq b,$$

and the 2-variable XNOR switching function is the inverse of the XOR function, so that

$$\text{XNOR}(a,b) = 1 \text{ if, and only if,} a = b.$$

Their symbols are shown in Fig. 2.21 and their truth tables are defined in Table 2.11.

The following algebraic symbols are used:

$$a \oplus b = \text{XOR}(a,b), a \equiv b = \text{XNOR}(a,b).$$

An equivalent definition of the XOR function is

$$\text{XOR}(a,b) = (a+b)\text{mod}2 = a \oplus b.$$

With this equivalent definition an n-variable XOR switching function can be defined for any $n > 2$:

$$\text{XOR}(a_1, a_2, \ldots, a_n) = (a_1 + a_2 + \ldots + a_n)\text{mod}2 = a_1 \oplus a_2 \oplus \ldots \oplus a_n,$$

and the n-variable XNOR switching function is the inverse of the XOR function:

$$\text{XNOR}(a_1, a_2, \ldots, a_n) = \overline{\text{XOR}(a_1,\ a_2,\ \ldots, a_n)}.$$

Examples of XOR gate and XNOR gate symbols are shown in Fig. 2.22.

Mod 2 sum is an associative operation, so that n-input XOR gates can be implemented with 2-input XOR gates. As an example, in Fig. 2.23a a 4-input XOR gate is implemented with three 2-input XOR gates.

An n-input XNOR gate is implemented by the same circuit as an n-input XOR gate in which the XOR gate that generates the output is substituted by an XNOR gate. In Fig. 2.23b a 4-input XNOR gate is implemented with two 2-input XOR gates and a 2-input XNOR gate.

XOR gates and XNOR gates are not universal modules. However they are very useful to implement arithmetic functions.

Example 2.5 As a first example consider a 4-bit magnitude comparator: given two 4-bit numbers $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$ generate a switching function *comp* equal to 1 if, and only if, $a = b$. The following trivial algorithm is used:

```
if (a3 = b3) and (a2 = b2) and (a1 = b1) and (a0 = b0)
then comp = 1;
else comp = 0;
end if;
```

The corresponding circuit is shown in Fig. 2.24: $comp = 1$ if, and only if, the four inputs of the NOR4 gate are equal to 0, that is, if $a_i = b_i$ and thus XOR(a_i, b_i) = 0, $\forall i = 0$, 1, 2, and 3.

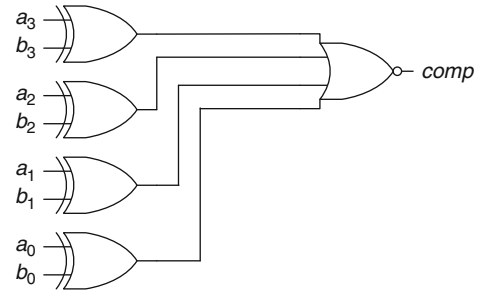


Fig. 2.24 4-Bit magnitude comparator

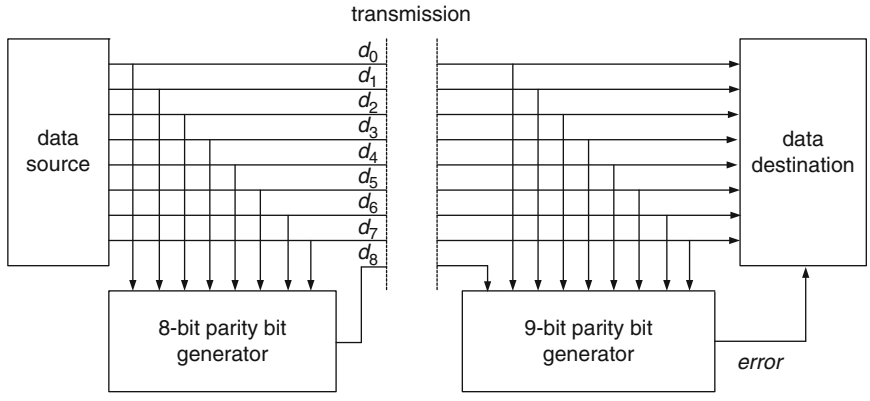


Fig. 2.25 Transmission of 8-bit data

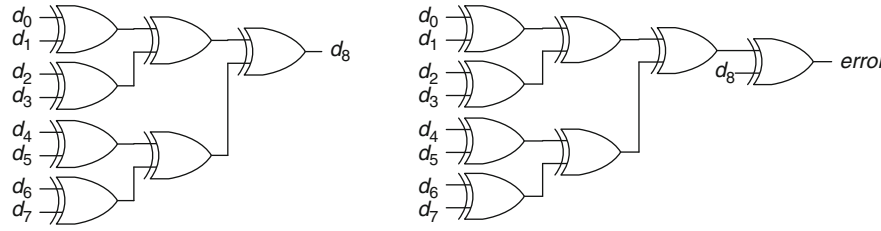
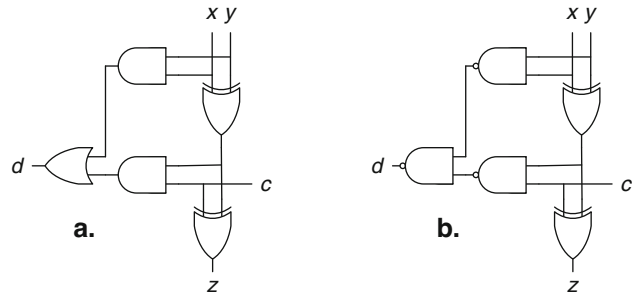


Fig. 2.26 Parity bit generation and parity check

Fig. 2.27 1-Bit adder



Example 2.6 The second example is a parity bit generator. It implements an n-variable switching function $parity(a_0, a_1, \ldots, a_{n-1}) = 1$ if, and only if, there is an odd number of 1s among variables a_0, $a_1, \ldots, a_{n-1}$. In other words,

$$parity(a_0, a_1, \ldots, a_{n-1}) = (a_0 + a_1 + \ldots + a_{n-1}) \bmod 2 = a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}.$$

Consider a communication system (Fig. 2.25) that must transmit 8-bit data $d = d_0 d_1 \ldots d_7$ from a data source circuit to a data destination circuit. On the source size, an 8-bit parity generator generates an additional bit $d_8 = d_0\ d_1\ \ldots\ d_7$, and the nine bits $d_0 d_1 \ldots d_7 d_8$ are transmitted. Thus, the number of 1s among the transmitted bits d_0, d_1, …, d_8 is always even. On the destination side, a 9-bit parity generator checks whether the number of 1s among d_0, d_1, …, d_8 is even, or not. If even, the parity generator output is equal to 0; if odd, the output is equal to 1. If it is assumed that during the transmission at most one bit could have been modified, due to the noise on the transmission lines, the 9-bit parity generator output is an *error* signal equal to 0 if no error has happened and equal to 1 in the contrary case.

An 8-bit parity generator and a 9-bit parity generator implemented with XOR2 gates are shown in Fig. 2.26.

Example 2.7 The most common use of XOR gates is within adders. A 1-bit adder implements two switching functions z and d defined by Table 2.3 and by (2.19) and (2.20). According to Table 2.3, z can also be expressed as follows:

$$z = (x + y + c) \bmod 2 = x \oplus y \oplus c. \tag{2.29}$$

On the other hand, d is equal to 1 if, and only if, $x + y + c \geq 2$. This condition can be expressed in the following way: either $x = y = 1$ or $c = 1$ and $x \neq y$. The corresponding Boolean expression is

Fig. 2.28 Tristate buffer and tristate inverter symbols

Table 2.12 Definition of tristate buffer and tristate inverter

$c\ x$	3-State buffer output y	3-State inverter output y
0 0	Z	Z
0 1	Z	Z
1 0	0	1
1 1	1	0

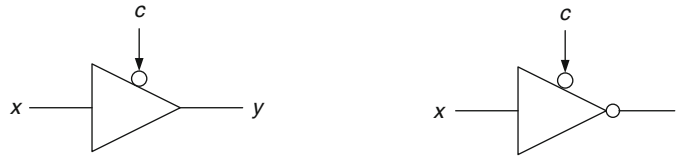


Fig. 2.29 Symbols of tristate components with active-low control input

Table 2.13 Definition of tristate components with active-low control input

$c\ x$	3-State buffer output y	3-State inverter output y
0 0	0	1
0 1	1	0
1 0	Z	Z
1 1	Z	Z

$$d = x \cdot y + c \cdot (x \oplus y). \tag{2.30}$$

The circuit that corresponds to (2.29) and (2.30) is shown in Fig. 2.27a. As mentioned above (Fig. 2.20), AND gates and OR gates can be implemented with NAND gates (Fig. 2.27b).

2.4.3 Tristate Buffers and Tristate Inverters

Tristate buffers and tristate inverters are components whose output can be in three different states: 0 (low voltage), 1 (high voltage), or Z (disconnected). A tristate buffer CMOS implementation is shown in Fig. 1.28a: when the control input $c = 0$, the output is disconnected from the input, so that the output impedance is very high (infinite if leakage currents are not considered); if $c = 1$, the output is connected to the input through a CMOS switch.

A tristate inverter is equivalent to an inverter whose output is connected to a tristate buffer. It works as follows: when the control input $c = 0$, the output is disconnected from the input; if $c = 1$, the output is equal to the inverse of the input.

The symbols of a tristate buffer and of a tristate inverter are shown in Fig. 2.28 and their working is defined in Table 2.12.

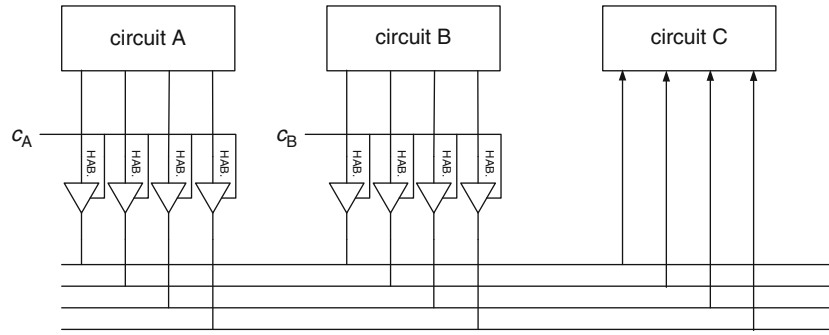


Fig. 2.30 4-Bit bus

Table 2.14 4-Bit bus definition

$c_A c_B$	Data transmission
0 0	None
0 1	B → C
1 0	A → C
1 1	Not allowed

In some tristate components the control signal c is active at low level. The corresponding symbols and definitions are shown in Fig. 2.29 and Table 2.13.

A typical application of tristate components is shown in Fig. 2.30. It is a 4-bit bus that permits to send 4-bit data either from circuit A to circuit C or from circuit B to circuit C. As an example, A could be a memory, B an input interface, and C a processor. Both circuits A and B must be able to send data to C but cannot be directly connected to C. To avoid collisions, 3-state buffers are inserted between A and B outputs and the set of wires connected to the circuit C inputs. To transmit data from A to C, $c_A = 1$ and $c_B = 0$, and to transmit data from B to C, $c_A = 0$ and $c_B = 1$ (Table 2.14).

2.5 Synthesis Tools

In order to efficiently implement combinational circuits, synthesis tools are necessary. In this section, some of the principles used to optimize combinational circuits are described.

2.5.1 Redundant Terms

When defining a switching function it might be that for some combinations of input variable values the corresponding output value is not defined because either those input value combinations never happen or because the function value does not matter. In the truth table, the corresponding entries are named “don’t care” (instead of 0 or 1). When defining a Boolean expression that describes the switching function to be implemented, the minterms that correspond to those don’t care entries can be used, or not, in order to optimize the final circuit.

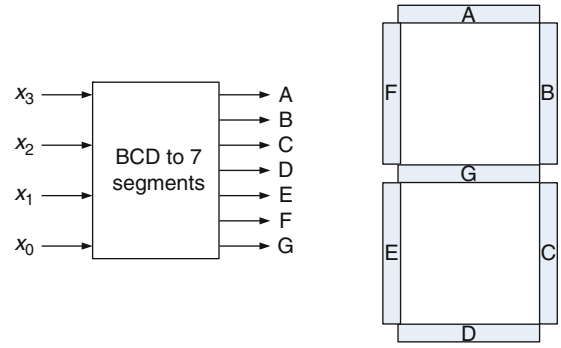


Fig. 2.31 BCD to 7-segment decoder

Table 2.15 BCD to 7-segment decoder definition

Digit	$x_3x_2x_1x_0$	A	B	C	D	E	F	G
0	0000	1	1	1	1	1	1	0
1	0001	0	1	1	0	0	0	0
2	0010	1	1	0	1	1	0	1
3	0011	1	1	1	1	0	0	1
4	0100	0	1	1	0	0	1	1
5	0101	1	0	1	1	0	1	1
6	0110	1	0	1	1	1	1	1
7	0111	1	1	1	0	0	0	0
8	1000	1	1	1	1	1	1	1
9	1001	1	1	1	0	0	1	1
	1010	–	–	–	–	–	–	–
	1011	–	–	–	–	–	–	–
	1100	–	–	–	–	–	–	–
	1101	–	–	–	–	–	–	–
	1110	–	–	–	–	–	–	–
	1111	–	–	–	–	–	–	–

Example 2.8 A BCD to 7-segment decoder (Fig. 2.31) is a combinational circuit with four inputs x_3, x_2, x_1, and x_0 that are the binary representation of a decimal digit (BCD means binary coded decimal) and seven outputs that control the seven segments of a display.

Among the 16 combinations of x_3, x_2, x_1, and x_0 values, only 10 are used: those that correspond to digits 0–9. Thus, the values of outputs A to G that correspond to inputs 1010 to 1111 are unspecified (don't care). The BCD to 7-segment decoder is defined by Table 2.15.

If all don't care entries are substituted by 0s, the following set of Boolean expressions is obtained:

$$\mathrm{A} = \bar{x}_3.x_1 + \bar{x}_3.x_2.x_0 + x_3.\bar{x}_2.\bar{x}_1, \tag{2.31a}$$

$$\mathrm{B} = \bar{x}_3.\bar{x}_2 + \bar{x}_2.\bar{x}_1 + \bar{x}_3.\bar{x}_1.\bar{x}_0 + \bar{x}_3.x_1.x_0, \tag{2.31b}$$

$$\mathrm{C} = \bar{x}_2.\bar{x}_1 + \bar{x}_3.x_0 + \bar{x}_3.\,x_2, \tag{2.31c}$$

$$\mathrm{D} = \bar{x}_2.\,\bar{x}_1.\bar{x}_0 + \bar{x}_3.\bar{x}_2.x_1 + \bar{x}_3.x_1.\bar{x}_0 + \bar{x}_3.x_2.\bar{x}_1.x_0, \tag{2.31d}$$

Table 2.16 Another definition of function B

$x_3x_2x_1x_0$	B
0000	1
0001	1
0010	1
0011	1
0100	1
0101	0
0110	0
0111	1
1000	1
1001	1
1010	1
1011	1
1100	1
1101	0
1110	0
1111	1

$$\mathrm{E} = \overline{x}_2.\overline{x}_1.\overline{x}_0 + \overline{x}_3.x_1.\overline{x}_0, \tag{2.31e}$$

$$\mathrm{F} = \overline{x}_3.\overline{x}_1.\overline{x}_0 + \overline{x}_3.x_2.\overline{x}_1 + \overline{x}_3.x_2.\overline{x}_0 + x_3.\overline{x}_2.\overline{x}_1, \tag{2.31f}$$

$$\mathrm{G} = \overline{x}_3.\overline{x}_2.x_1 + \overline{x}_3.x_2.\overline{x}_1 + \overline{x}_3.x_2.\overline{x}_0 + x_3.\overline{x}_2.\overline{x}_1. \tag{2.31g}$$

For example, B can be expressed as the sum of minterms m_0, m_1, m_2, m_3, m_4, m_7, m_8, and m_9:

$$\begin{aligned}\mathrm{B} = {} & \overline{x_3}.\overline{x_2}.\overline{x_1}.\overline{x_0} + \overline{x_3}.\overline{x_2}.\overline{x_1}.x_0 + \overline{x_3}.\overline{x_2}.x_1.\overline{x_0} + \overline{x_3}.\overline{x_2}.x_1.x_0 + \\ & \overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1}.\overline{x_0} + x_3.\overline{x_2}.\overline{x_1}.x_0.\end{aligned}$$

Then, the previous expression can be minimized:

$$\begin{aligned}\mathrm{B} &= \overline{x_3}.\overline{x_2}.(\overline{x_1}.\overline{x_0} + \overline{x_1}.x_0 + x_1.\overline{x_0} + \overline{x_1}.\overline{x_0}) + \overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \\ &\quad \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1}.(\overline{x_0} + x_0) \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1} \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.\overline{x_2}.\overline{x_1}.\overline{x_0} + \overline{x_3}.\overline{x_2}.x_1.x_0 + \overline{x_3}.\overline{x_2}.\overline{x_1} + \overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \\ &\quad \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1} \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.(\overline{x_2} + x_2).\overline{x_1}.\overline{x_0} + \overline{x_3}.(\overline{x_2} + x_2).x_1.x_0 + (\overline{x_3} + x_3).\overline{x_2}.\overline{x_1} \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.\overline{x_1}.\overline{x_0} + \overline{x_3}.x_1.x_0 + \overline{x_2}.\overline{x_1}.\end{aligned} \tag{2.32}$$

By performing the same type of optimization for all other functions, the set of (2.31) has been obtained.

In Table 2.16 the don't care entries of function B have been defined in another way and a different Boolean expression is obtained: according to Table 2.16, B = 1 if, and only if, $x_2 = 0$ or $x_1x_0 = 00$ or 11; thus

$$\mathrm{B} = \overline{x}_2 + \overline{x}_1.\overline{x}_0 + x_1.x_0. \tag{2.33}$$

Equations 2.32 and 2.33 are compatible with the initial specification (Table 2.15). They generate different values of B when $x_3x_2x_1x_0 = 1010$, 1011, 1100, or 1111, but in those cases the value of B

Table 2.17 Comparison between (2.31) and (2.34)

Gate type	Number of gates (2.31)	Number of gates (2.34)
AND2	6	14
AND3	17	1
AND4	1	-
OR2	1	1
OR3	2	2
OR4	4	4
NOT	4	4

does not matter. On the other hand (2.33) is simpler than (2.32) and would correspond to a better implementation.

By performing the same type of optimization for all other functions, the following set of expressions has been obtained:

$$\mathrm{A} = x_1 + x_2.x_0 + x_3, \tag{2.34a}$$

$$\mathrm{B} = \overline{x}_2 + \overline{x}_1.\overline{x}_0 + x_1.x_0, \tag{2.34b}$$

$$\mathrm{C} = \overline{x}_1 + x_0 + x_2, \tag{2.34c}$$

$$\mathrm{D} = \overline{x}_2.\overline{x}_0 + \overline{x}_2.x_1 + x_1.\overline{x}_0 + x_2.\overline{x}_1.x_0, \tag{2.34d}$$

$$\mathrm{E} = \overline{x}_2.\overline{x}_0 + x_1.\overline{x}_0, \tag{2.34e}$$

$$\mathrm{F} = \overline{x}_1.\overline{x}_0 + x_2.\overline{x}_1 + x_2.\overline{x}_0 + x_3, \tag{2.34f}$$

$$\mathrm{G} = \overline{x}_2.\overline{x}_0 + x_2.\overline{x}_1 + x_1.\overline{x}_0 + x_3. \tag{2.34g}$$

To summarize:

- If the "don't care" of Table 2.15 are replaced by 0s, the set of (2.31) is obtained.
- If they are replaced by either 0 or 1, according to some optimization method (not described in this course), the set of (2.34) would have been obtained.

In Table 2.17 the numbers of AND, OR, and NOT gates necessary to implement (2.31) and (2.34) are shown.

The circuit that implements (2.31) has $6 \cdot 2 + 17 \cdot 3 + 1 \cdot 4 + 1 \cdot 2 + 2 \cdot 3 + 4 \cdot 4 + 4 \cdot 1 = 95$ gate inputs and the circuit that implements (2.34) has $14 \cdot 2 + 1 \cdot 3 + 1 \cdot 2 + 2 \cdot 3 + 4 \cdot 4 + 4 \cdot 1 = 59$ gate inputs. Obviously, the second circuit is better.

2.5.2 Cube Representation

A combinational circuit synthesis tool is a set of programs that generates optimized circuits, according to some criteria (cost, delay, power) starting either from logic expressions or from tables. The cube representation of combinational functions is an easy way to define Boolean expressions within a computer programming environment.

The set of n-component binary vectors ${B_2}^n$ can be considered as a cube (actually a hypercube) of dimension n. For example, if $n = 3$, the set ${B_2}^3$ of 3-component binary vectors is represented by the cube of Fig. 2.32a.

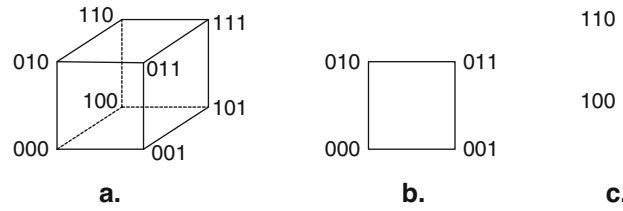


Fig. 2.32 Cubes

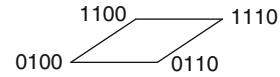


Fig. 2.33 Solutions of $x_2 \cdot \overline{x_0} = 1$

A subset of ${B_2}^n$ defined by giving a particular value to m vector components is a subcube ${B_2}^{n-m}$ of dimension n-m. As an example, the subset of vectors of ${B_2}^3$ whose first coordinate is equal to 0 (Fig. 2.32b) is a cube of dimension 2 (actually a square). Another example: the subset of vectors of ${B_2}^3$ whose first coordinate is 1 and whose third coordinate is 0 (Fig. 2.32c) is a cube of dimension 1 (actually a straight line).

Consider a 4-variable function f defined by the following Boolean expression:

$$f(x_3, x_2, x_1, x_0) = x_2 \cdot \overline{x_0}.$$

This function is equal to 1 if, and only if, $x_2 = 1$ and $x_0 = 0$, that is

$$f = 1 \text{ iff } (x_3, x_2, x_1, x_0) \in \left\{ x \in {B_2}^4 \middle| x_2 = 1 \text{ and } x_0 = 0 \right\}.$$

In other words, $f = 1$ if, and only if, (x_3, x_2, x_1, x_0) belongs to the 2-dimensional cube of Fig. 2.33.

This example suggests another definition.

Definition 2.2 A cube is a set of elements of ${B_2}^n$ where a product of literals (Definitions 2.1) is equal to 1.

In this chapter switching functions have been expressed under the form of sums of products of literals (e.g., (2.19) and (2.20)), and to those expressions correspond implementations by means of logic gates (e.g., Figs. 2.17 and 2.16). According to Definition 2.2, a set of elements of ${B_2}^n$ where a product of literals is equal to 1 is a cube. Thus, a sum of product of literals can also be defined as a union of cubes that defines the set of points of ${B_2}^n$ where $f = 1$. In what follows cube and product of literals are considered as synonymous.

How can a product of literals be represented within a computer programming environment? For that an order of the variables must be defined, for example (as above) x_{n-1}, x_{n-2}, …, x_1, x_0. Then, consider a product p of literals. It is represented by an n-component ternary vector ($p_{n-1}, p_{n-2}, \ldots, p_1$, p_0) where

- $p_i = 0$ if x_i is in p under inverted form ($\overline{x_i}$).
- $p_i = 1$ if x_i is in p under non-inverted form (x_i).
- $p_i = \mathrm{X}$ if x_i is not in p.

Example 2.9 (with $n = 4$) The set of cubes that describes (2.31d) is {X000, 001X, 0X10, 0101}, and the set of cubes that corresponds to (2.34g) is {X0X0, X10X, XX10, 1XXX}. Conversely, the product of literals represented by 1X01 is $x_3 \cdot \overline{x_1} \cdot x_0$ and the product of literals represented by X1X0 is $x_2 \cdot \overline{x_0}$.

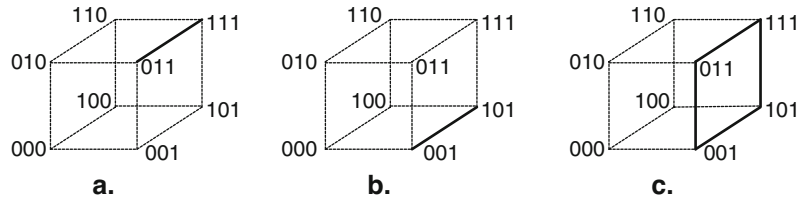


Fig. 2.34 Union of cubes

2.5.3 Adjacency

Adjacency is the basic concept that permits to optimize Boolean expressions. Two m-dimensional cubes are adjacent if their associated ternary vectors differ in only one position. As an example ($n = 3$), the 1-dimensional cubes X11 (Fig. 2.34a) and X01 (Fig. 2.34b) are adjacent and their union is a 2-dimensional cube XX1 = X11 ∪ X01 (Fig. 2.34c).

The corresponding products of literals are the following: X11 represents $x_1 \cdot x_0$, X01 represents $\overline{x_1} \cdot x_0$, and their union XX1 represents x_0. In terms of products of literals, the union of the two adjacent cubes is the sum of the corresponding products:

$$x_1 \cdot x_0 + \overline{x_1} \cdot x_0 = (x_1 + \overline{x_1}) \cdot x_0 = 1 \cdot x_0 = x_0.$$

Thus, if a function f is defined by a union of cubes and if two cubes are adjacent, then they can be replaced by their union. The result, in terms of products of literals, is that two products of n-m literals are replaced by a single product of $n - m - 1$ literals.

Example 2.10 A function f of four variables a, b, c, and d is defined by its minterms (Definition 2.1):

$$f(a,b,c,d) = \overline{a} \cdot \overline{b} \cdot c \cdot \overline{d} + \overline{a} \cdot \overline{b} \cdot c \cdot d + \overline{a} \cdot b \cdot \overline{c} \cdot d + \overline{a} \cdot b \cdot c \cdot \overline{d} + \overline{a} \cdot b \cdot c \cdot d + a \cdot \overline{b} \cdot \overline{c} \cdot \overline{d}.$$

The corresponding set of cubes is

$$\{0010,\ 0011,\ 0101,\ 0110,\ 0111,\ 1000\}.$$

The following adjacencies permit to simplify the representation of f:

$$0010 \cup 0011 = 001\text{X},$$
$$0110 \cup 0111 = 011\text{X},$$
$$0101 \cup 0111 = 01\text{X}1.$$

Thanks to the idempotence property (2.9) the same cube (0111 in this example) can be used several times. The simplified set of cubes is

$$\{001\text{X},\ 011\text{X},\ 01\text{X}1,\ 1000\}.$$

There remains an adjacency:

$$001\text{X} \cup 011\text{X} = 0\text{X}1\text{X}.$$

The final result is

$$\{0X1X,\ 01X1,\ 1000\}$$

and the corresponding Boolean expression is

$$f = \overline{a} \cdot c + \overline{a} \cdot b \cdot d + a \cdot \overline{b} \cdot \overline{c} \cdot \overline{d}.$$

To conclude, a repeated use of the fact that two adjacent cubes can be replaced by a single cube permits to generate new Boolean expressions, equivalent to the initial one and with fewer terms. Furthermore the new terms have fewer literals. This is the basis of most automatic optimization tools.

All commercial synthesis tools include programs that automatically generate optimal circuits according to some criteria such as cost, delay, or power consumption, and starting from several types of specification. For education purpose open-source tools are available, for example C. Burch (2005).

2.5.4 Karnaugh Map

In the case of switching functions of a few variables, a graphical method can be used to detect adjacencies and to optimize Boolean expressions. Consider the function $f(a, b, c, d)$ of Example 2.10. It can be represented by the Karnaugh map (Karnaugh 1953) of Fig. 2.35a. Observe the enumeration ordering of rows and columns (00, 01, 11, 10): the variable values that correspond to a row (a column) and to the next row (the next column) differ in only one position.

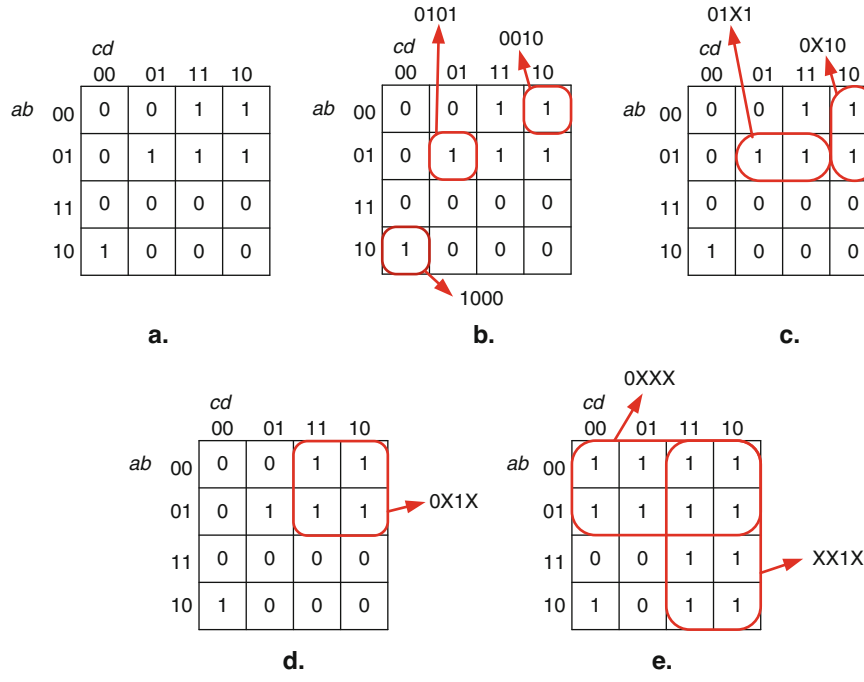


Fig. 2.35 Karnaugh maps

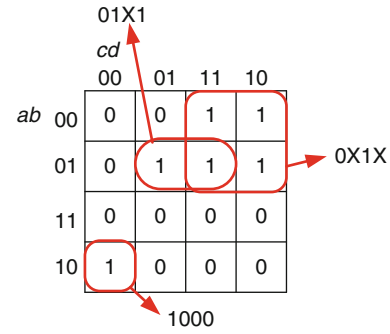


Fig. 2.36 Optimization of f

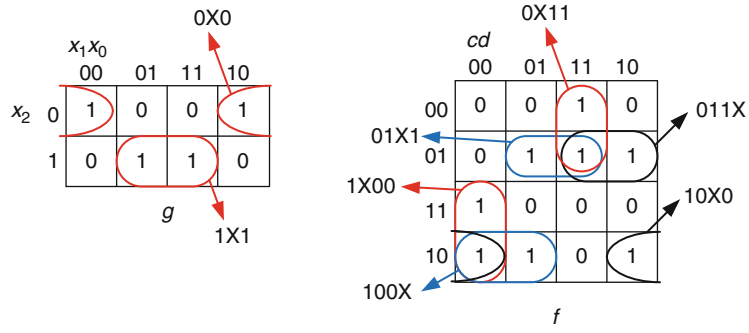


Fig. 2.37 Functions g and h

To each one of this graphical representation is associated a minterm of the function (a 0-dimensional cube). Several examples are shown in Fig. 2.35b. Thanks to the chosen enumeration ordering, to groups of two adjacent 1s like those of Fig. 2.35c are associated 1-dimensional cubes. To a group of four adjacent 1s like the one of Fig. 2.35d is associated a 2-dimensional cube. To groups of eight adjacent 1s like those of Fig. 2.35e (another switching function) are associated 3-dimensional cubes.

Thus (Fig. 2.36) the function $f(a, b, c, d)$ of Example 2.10 can be expressed as the Boolean sum of three cubes 0X1X, 01X1, and 1000 so that

$$f = \overline{a} \cdot c + \overline{a} \cdot b \cdot d + a \cdot \overline{b} \cdot \overline{c} \cdot \overline{d}.$$

It is important to observe that the rightmost cells and the leftmost cells are adjacent, and so are also the uppermost cells and the downmost cells (as if the map were drawn on the surface of a torus).

Two additional examples are given in Fig. 2.37. Function g of Fig. 2.37a can be expressed as the Boolean sum of two 1-dimensional cubes 0X0 and 1X1, so that

$$g = \overline{x_2} \cdot \overline{x_0} + x_2 \cdot x_0,$$

and function h of Fig. 2.37b can be expressed as the Boolean sum of six 1-dimensional cubes 01X1, 011X, 0X11, 10X0, 100X, and 1X00, so that

$$h = \overline{a} \cdot b \cdot d + \overline{a} \cdot b \cdot c + \overline{a} \cdot c \cdot d + a \cdot \overline{b} \cdot \overline{d} + a \cdot \overline{b} \cdot \overline{c} + a \cdot \overline{c} \cdot \overline{d}.$$

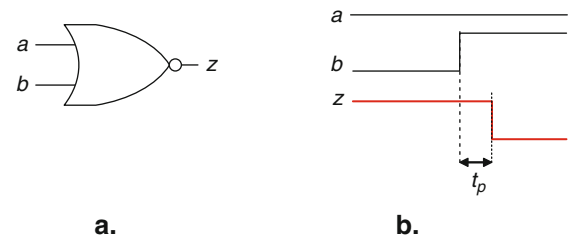


Fig. 2.38 Propagation time t_p

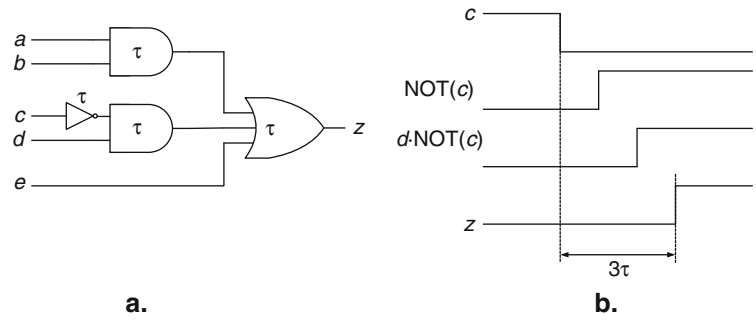


Fig. 2.39 Example of propagation time computation

2.6 Propagation Time

Logic components such as gates are physical systems. Any change of their state, for example the output voltage transition from some level to another level, needs some quantity of energy and therefore some time (zero delay would mean infinite power). Thus, apart from their function (AND2, OR3, NAND4, and so on), logic gates are also characterized by their propagation time (delay) between inputs and outputs.

Consider a simple NOR2 gate (Fig. 2.38a). Assume that initially $a = b = 0$. Then $z = \text{NOR}(0, 0) = 1$ (Fig. 2.38b). When b rises from 0 to 1, then $\text{NOR}(0, 1) = 0$ and z must fall from 1 to 0. However the output state change is not immediate; there is a small delay t_p generally expressed in nanoseconds (ns) or picoseconds (ps).

Example 2.11 The circuit of Fig. 2.39a implements a 5-variable switching function $z = a \cdot b + \overline{c} \cdot d + e$. Assume that all components (AND2, NOT, OR3) have the same propagation time τ ns. Initially $a = 0$, b is either 0 or 1, $c = 1$, $d = 1$, and $e = 0$. Thus $z = 0 \cdot b + \overline{1} \cdot 1 + 0 = 0$. If c falls from 1 down to 0 then the new value of z must be $z = 0 \cdot b + \overline{0} \cdot 1 + 0 = 1$. However this output state change takes some time: the inverter output $\overline{c}$ changes after τ ns; the AND2 output $\overline{c} \cdot d$ changes after 2τ ns, and the OR3 output z changes after 3τ ns (Fig. 2.39b).

Thus, the propagation time of a circuit depends on the component propagation times but also on the circuit itself. Two different circuits could implement the same switching circuit but with different propagation times.

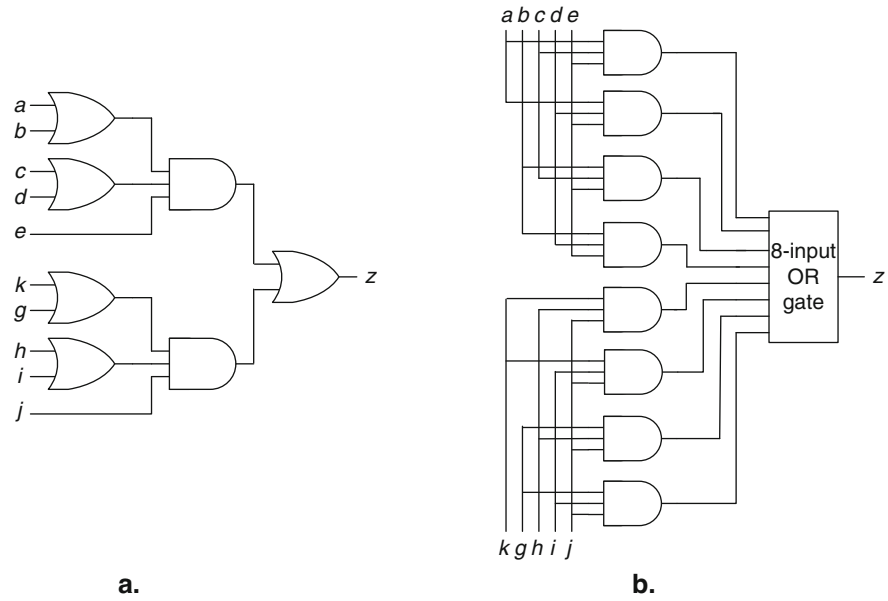
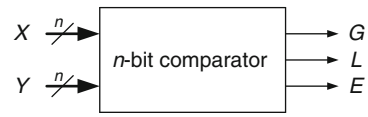


Fig. 2.40 Two circuits that implement the same function f

Fig. 2.41 n-Bit comparator



Example 2.12 The two following expressions define the same switching function z:

$$z = (a + b) \cdot (c + d) \cdot e + (k + g) \cdot (h + i) \cdot j,$$
$$z = a \cdot c \cdot e + a \cdot d \cdot e + b \cdot c \cdot e + b \cdot d \cdot e + k \cdot h \cdot j + k \cdot i \cdot j + g \cdot h \cdot j + g \cdot i \cdot j.$$

The corresponding circuits are shown in Fig. 2.40a, b. The circuit of Fig. 2.40a has 7 gates and 16 gate inputs while the circuit of Fig. 2.40b has 9 gates and 32 gate inputs. On the other hand, if all gates are assumed to have the same propagation time τ ns, then the circuit of Fig. 2.40a has a propagation time equal to 3τ ns while the circuit of Fig. 2.40b has a propagation time equal to 2τ ns. Thus, the circuit of Fig. 2.40a could be less expensive in terms of number of transistors but with a longer propagation time than the circuit of Fig. 2.40b. In function of the system specification, the designer will have to choose between a faster but more expensive implementation or a slower and cheaper implementation (speed vs. cost balance).

A more realistic example is now presented. An n-bit comparator (Fig. 2.41) is a circuit with two n-bit inputs $X = x_{n-1}x_{n-2} \ldots x_0$ and $Y = y_{n-1}y_{n-2} \ldots y_0$ that represent two naturals and three 1-bit outputs G (greater), L (lower), and E (equal). It works as follows: $G = 1$ if $X > Y$, otherwise $G = 0$; $L = 1$ if $X < Y$, otherwise $L = 0$; $E = 1$ if $X = Y$, otherwise $E = 0$.

A step-by-step algorithm can be used. For that, the pairs of bits (x_i, y_i) are sequentially explored starting from the most significant bits (x_{n-1}, y_{n-1}). Initially $G = 0$, $L = 0$, and $E = 1$. As long as $x_i = y_i$, the values of G, L, and E do not change. When for the first time $x_i \neq y_i$, there are two possibilities: if $x_i > y_i$ then $G = 1, L = 0$, and $E = 0$, and if if $x_i < y_i$ then $G = 0, L = 1$, and $E = 0$. From this step, the values of G, L, and E do not change any more.

Table 2.18 Magnitude comparison

X	1	0	1	1	0 or 1	0 or 1	0 or 1	0 or 1
Y	1	0	1	0	0 or 1	0 or 1	0 or 1	0 or 1
G	0	0	0	1	1	1	1	1
L	0	0	0	0	0	0	0	0
E	1	1	1	0	0	0	0	0

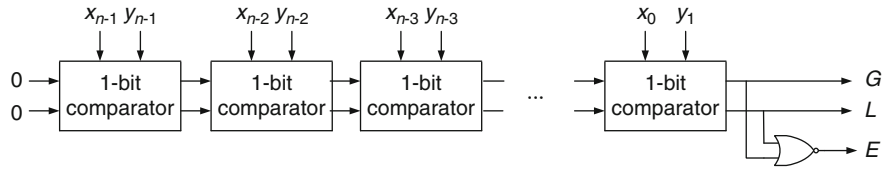


Fig. 2.42 Comparator structure

Algorithm 2.2 Magnitude Comparison

```
G = 0; L = 0; E = 1;
for i in n-1 downto 0 loop
  if E = 1 and x_i > y_i then
     G = 1; L = 0; E = 0;
  elsif E = 1 and x_i < y_i then
     G = 0; L = 1; E = 0;
  end if;
end loop;
```

This method is correct because in binary the weight of bits x_i and y_i is 2^i and is greater than $2^{i-1} + 2^{i-2} + \ldots + 2^0 = 2^i - 1$. An example of computation is given in Table 2.18 with $n = 8$, $X = 1011\text{----}$ and $Y = 1010\text{----}$.

The corresponding circuit structure is shown in Fig. 2.42. Obviously $E = 1$ if $G = 0$ and $L = 0$ so that $E = \text{NOR}\ (G, L)$.

Every block (Fig. 2.43) executes the loop body of Algorithm 2.2 and is defined by the following Boolean expressions where $E_i = \overline{G_i} \cdot \overline{L_i}$:

$$G_{i-1} = E_i \cdot x_i \cdot \overline{y_i} + \overline{E_i} \cdot G_i = \overline{G_i} \cdot \overline{L_i} \cdot x_i \cdot \overline{y_i} + (G_i + L_i) \cdot G_i = \overline{L_i} \cdot x_i \cdot \overline{y_i} + G_i, \tag{2.35}$$

$$L_{i-1} = E_i \cdot \overline{x_i} \cdot y_i + \overline{E_i} \cdot L_i = \overline{G_i} \cdot \overline{L_i} \cdot \overline{x_i} \cdot y_i + (G_i + L_i) \cdot L_i = \overline{G_i} \cdot \overline{x_i} \cdot y_i + L_i \tag{2.36}$$

The circuit that implements (2.35) and (2.36) is shown in Fig. 2.44. It contains 8 gates (including the inverters) and 14 gate inputs, and the propagation time is 3τ ns assuming as before that all components (NOT, AND3, and OR2) have the same delay τ ns.

The complete n-bit comparator (Fig. 2.42) contains $8n + 1$ gates and $14n + 2$ gate inputs and has a propagation time equal to $(3n + 1)\tau$ ns.

Instead of reading the bits of X and Y one at a time, consider an algorithm that reads two bits of X and Y at each step. Assume that $n = 2\ m$. Then the following Algorithm 2.3 is similar to Algorithm 2.2. The difference is that two successive bits x_{2j+1} and x_{2j} of X and two successive bits y_{2j+1} and y_{2j} of Y are considered. Those pairs of bits can be interpreted as quaternary digits (base-4 digits).

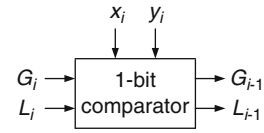


Fig. 2.43 1-Bit comparator

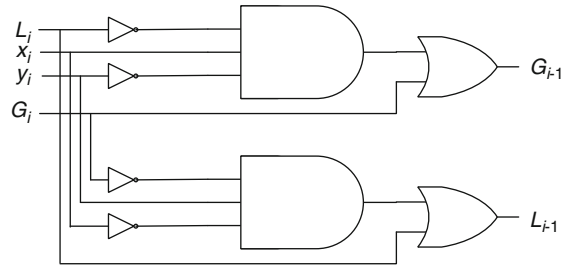


Fig. 2.44 1-Bit comparator implementation

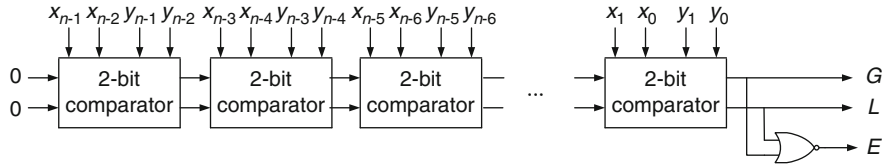


Fig. 2.45 Comparator structure (version 2)

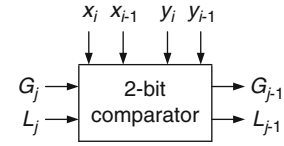


Fig. 2.46 2-Bit comparator

Algorithm 2.3 Magnitude Comparison, Version 2

```
G = 0; L = 0; E = 1;
for j in m-1 downto 0 loop
  if E = 1 and x2j+1 2j > y2j+1 2jthen
     G = 1; L = 0; E = 0;
  elsif E = 1 and x2j+1 2j < y2j+1 2jthen
     G = 0; L = 1; E = 0;
  end if;
end loop;
```

The corresponding circuit structure is shown in Fig. 2.45.

Every block (Fig. 2.46) executes the loop body of Algorithm 2.3 and is defined by Table 2.19 to which correspond the following equations:

$$G_{j-1} = \overline{L_j} \cdot x_{i-1} \cdot \overline{y_i} \cdot \overline{y_{i-1}} + \overline{L_j} \cdot x_i \cdot \overline{y_i} + \overline{L_j} \cdot x_i \cdot x_{i-1} \cdot \overline{y_{i-1}} + G_j, \tag{2.37}$$

$$L_{j-1} = \overline{G_j} \cdot y_{i-1} \cdot \overline{x_i} \cdot \overline{x_{i-1}} + \overline{G_j} \cdot y_i \cdot \overline{x_i} + \overline{G_j} \cdot y_i \cdot y_{i-1} \cdot \overline{x_{i-1}} + L_j, \tag{2.38}$$

where $i = 2j + 1$.

Table 2.19 2-Bit comparator definition

G_j	L_j	x_i	x_{i-1}	y_i	y_{i-1}	G_{j-1}	L_{j-1}
0	0	0	0	0	0	0	0
0	0	0	0	1	–	0	1
0	0	0	0	–	1	0	1
0	0	0	1	0	0	1	0
0	0	0	1	0	1	0	0
0	0	0	1	1	–	0	1
0	0	1	0	0	–	1	0
0	0	1	0	1	0	0	0
0	0	1	0	1	1	0	1
0	0	1	1	0	–	1	0
0	0	1	1	–	0	1	0
0	0	1	1	1	1	0	0
0	1	–	–	–	–	0	1
1	0	–	–	–	–	1	0
1	1	–	–	–	–	–	–

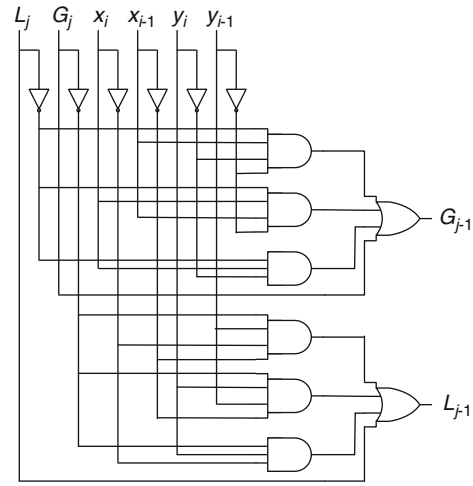


Fig. 2.47 2-Bit comparator implementation

Table 2.20 Comparison between the circuits of Figs. 2.42 and 2.45

Circuit	Gates	Gate inputs	Propagation time
Figure 2.42	$8n + 1$	$14n + 2$	$(3n + 1)\tau$
Figure 2.45	$7n + 1$	$18n + 2$	$(1.5n + 1)\tau$

The circuit that implements (2.37) and (2.38) is shown in Fig. 2.47. It contains 14 gates (including the inverters) and 36 gate inputs, and the propagation time is 3τ ns assuming as before that all components (NOT, AND3, AND4, and OR4) have the same delay τ ns.

The complete n-bit comparator (Fig. 2.45), with $n = 2m$, contains $14m + 1 = 7n + 1$ gates and $36m + 2 = 18n + 2$ gate inputs and has a propagation time equal to $(3m + 1)\tau = (1.5n + 1)\tau$ ns.

To summarize (Table 2.20) the circuit of Fig. 2.45 has fewer gates, more gate inputs, and a shorter propagation time than the circuit of Fig. 2.42 (roughly half the propagation time).

2.7 Other Logic Blocks

Apart from the logic gates, some other components are available and can be used to implement combinational circuits.

2.7.1 Multiplexers

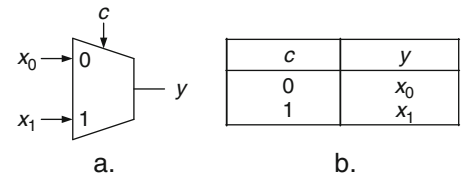
The circuit of Fig. 2.48a is a 1-bit 2-to-1 multiplexer (MUX2-1). It has two data inputs x_0 and x_1, a control input c, and a data output y. It works as follows (Fig. 2.48b): when $c = 0$ the data output y is connected to the data input x_0 and when $c = 1$ the data output y is connected to the data input x_1. So, the main function of a multiplexer is to implement controllable connections.

A typical application is shown in Fig. 2.49: the input of *circuit_C* can be connected to the output of either *circuit_A* or *circuit_B* under the control of signal *control*:

- If *control* = 0, *circuit_C* input = *circuit_A* output.
- If *control* = 1, *circuit_C* input = *circuit_B* output.

More complex multiplexers can be defined. An m-bit 2^n-to-1 multiplexer has $2^n m$-bit data inputs $x_0, x_1, \ldots, x_{2^n-1}$, an n-bit control input c, and an m-bit data output y. It works as follows: if c is equal to the binary representation of natural i, then $y = x_i$.

Two examples are given in Fig. 2.50: in Fig. 2.50a the symbol of an m-bit 2-to-1 multiplexer is shown, and in Fig. 2.50b the symbol and the truth table of a 1-bit 4-to-1 multiplexer (MUX4-1) are shown.



c	y
0	x_0
1	x_1

Fig. 2.48 1-Bit 2-to-1 multiplexer

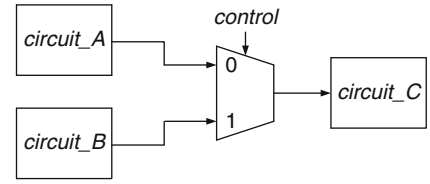
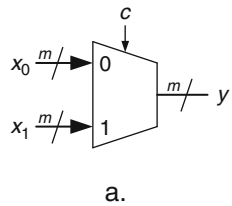
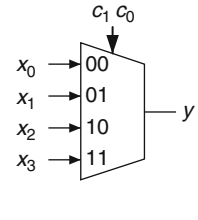


Fig. 2.49 Example of controllable connection





c_1 c_0	y
00	x_0
01	x_1
10	x_2
11	x_3

Fig. 2.50 Examples of multiplexers

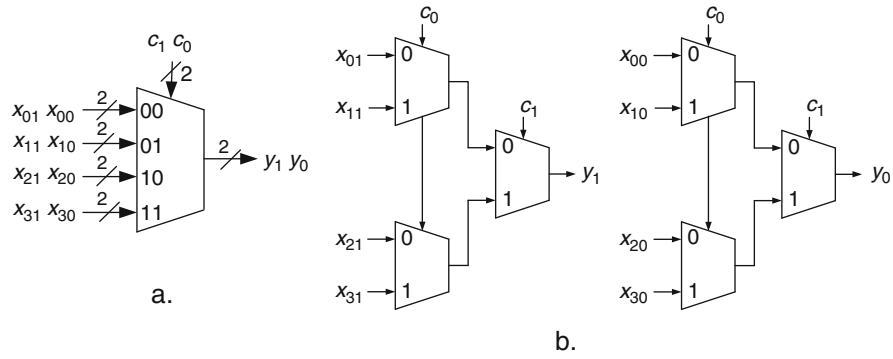
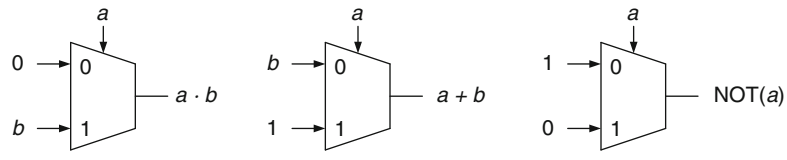


Fig. 2.51 2-Bit MUX4-1 implemented with six 1-bit MUX2-1

Fig. 2.52 MUX2-1 is a universal module



x_2	x_1	x_0	y_1	y_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

a.

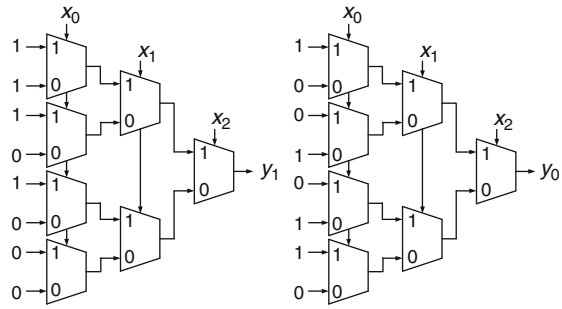


Fig. 2.53 Implementation of two 3-variable switching functions

In fact, any multiplexer can be built with 1-bit 2-to-1 multiplexers. For example, Fig. 2.51a is the symbol of a 2-bit MUX4-1 and Fig. 2.51b is an implementation consisting of six 1-bit MUX2-1.

Multiplexers can also be used to implement switching functions. The function executed by the 1-bit MUX2-1 of Fig. 2.48 is

$$y = \overline{c} \cdot x_0 + c \cdot x_1. \tag{2.39}$$

In particular, MUX2-1 is a universal module (Fig. 2.52):

- If $c = a$, $x_0 = 0$ and $x_1 = b$, then $y = a \cdot b$.
- If $c = a$, $x_0 = b$ and $x_1 = 1$, then $y = \overline{a} \cdot b + a = a + b$.
- If $c = a$, $x_0 = 1$ and $x_1 = 0$, then $y = \overline{a}$.

Furthermore, any switching function of n variables can be implemented by a 2^n-to-1 multiplexer. As an example, consider the 3-variable switching functions y_1 and y_0 of Fig. 2.53a. Each of them can be implemented by a MUX8-1 that in turn can be synthesized with seven MUX2-1 (Fig. 2.53b). The

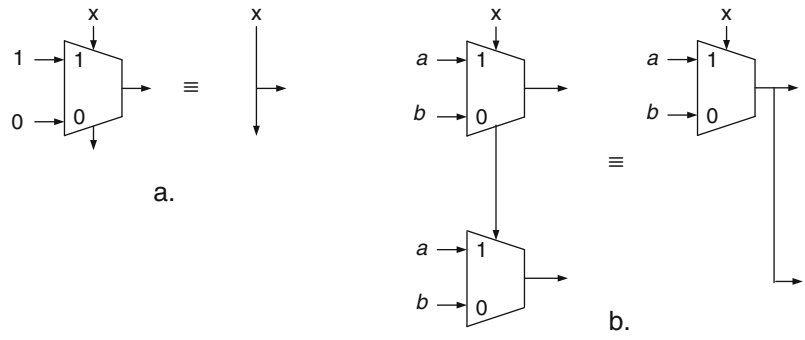


Fig. 2.54 Optimization rules

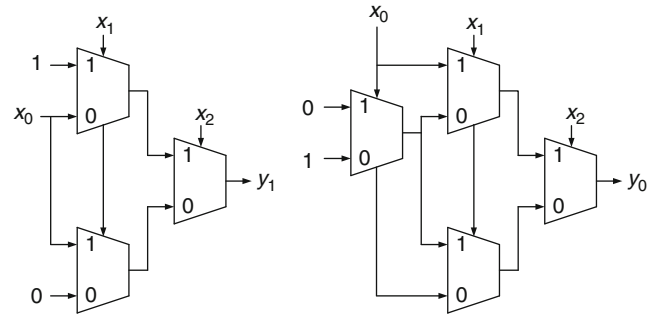


Fig. 2.55 Optimized circuits

three variables x_2, x_1, and x_0 are used to control the connection of the output (y_1 or y_0) to a constant value as defined in the function truth table.

In many cases the circuit can be simplified using simple and obvious rules. Two optimization rules are shown in Fig. 2.54. In Fig. 2.54a if $x = 0$ then the multiplexer output is equal to 0 and if $x = 1$ then the multiplexer output is equal to 1. Thus the multiplexer output is equal to x. In Fig. 2.54b two multiplexers controlled by the same variable x and with the same data inputs can be replaced by a unique multiplexer.

An optimized version of the circuits of Fig. 2.53b is shown in Fig. 2.55.

Two switching function synthesis methods using multiplexers have been described. The first is to use multiplexers to implement the basic Boolean operations (AND, OR, NOT), which is generally not a good idea, rather a way to demonstrate that MUX2-1 is a universal module. The second is the use of an m-bit 2^n-to-1 multiplexer to implement m functions of n variables. In fact, an m-bit 2^n-to-1 multiplexer with all its data inputs connected to constant values implements the same function as a ROM storing $2^n m$-bit words. Then the 2^n-to-1 multiplexers can be synthesized with MUX2-1 and the circuits can be optimized using rules such as those of Fig. 2.54.

A more general switching function synthesis method with MUX2-1 components is based on (2.39) and on the fact that any n-variable switching function $f(x_0, x_1, \ldots, x_{n-1})$ can be expressed under the form

$$f(x_0, x_1, \ldots, x_{n-1}) = \overline{x_0} \cdot f_0(x_1, \ldots, x_{n-1}) + x_0 \cdot f_1(x_1, \ldots, x_{n-1}) \tag{2.40}$$

where

$$f_0(x_1, \ldots, x_{n-1}) = f(0, x_1, \ldots, x_{n-1}) \text{ and } f_1(x_1, \ldots, x_{n-1}) = f(1, x_1, \ldots, x_{n-1}) \tag{2.41}$$

are functions of $n - 1$ variables. The circuit of Fig. 2.56 is a direct consequence of (2.40) and (2.41). In this way variable x_0 has been extracted.

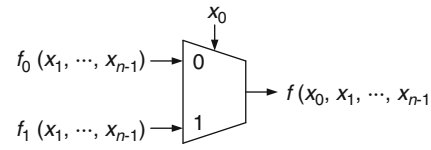


Fig. 2.56 Extraction of variable x_0

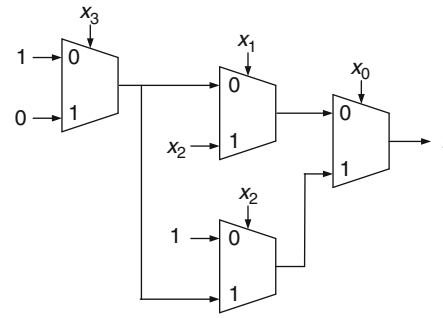


Fig. 2.57 MUX2-1 implementation of f

Then a similar variable extraction can be performed with functions f_0 and f_1 (not necessarily the same variable) so that functions of $n-2$ variables are obtained, and so on. Thus, an iterative extraction of variables finally generates constants (0-variable functions), variables, or already generated functions.

Example 2.13 Use the variable extraction method to implement the following 4-variable function:

$$f = \overline{x_0} \cdot \overline{x_1} \cdot \overline{x_3} + \overline{x_0} \cdot x_1 \cdot x_2 + x_0 \cdot \overline{x_2} + x_0 \cdot \overline{x_3}.$$

First extract x_0: $f_0 = \overline{x_1} \cdot \overline{x_3} + x_1 \cdot x_2$ and $f_1 = \overline{x_2} + \overline{x_3}$.
Then extract x_1 from f_0: $f_{00} = \overline{x_3}$ and $f_{01} = x_2$.
Extract x_2 from f_1: $f_{10} = 1$ and $f_{11} = \overline{x_3}$.
It remains to synthesize $\overline{x_3} = \overline{x_3} \cdot 1 + x_3 \cdot 0$.
The circuit is shown in Fig. 2.57.

2.7.2 Multiplexers and Memory Blocks

ROM blocks can be used to implement switching functions defined by their truth table (Sect. 2.2) but in most cases it is a very inefficient method. However the combined use of small ROM blocks, generally called LUT, and of multiplexers permits to define efficient circuits. This is a commonly used technique in field programmable devices such as FPGAs (Chap. 7).

Assume that 6-input LUTs (LUT6) are available. Then the variable extraction method of Fig. 2.56 can be iteratively applied up to the step where all obtained functions depend on at most six variables. As an example, the circuit of Fig. 2.58 implements any function of eight variables.

Observe that the rightmost part of the circuit of Fig. 2.58 synthesizes a function of six variables: x_6, x_7 and the four LUT6 outputs. An alternative circuit consisting of five LUT6 is shown in Fig. 2.59.

Figure 2.59 suggests a variable extraction method in which two variables are extracted at each step. It uses the following relation:

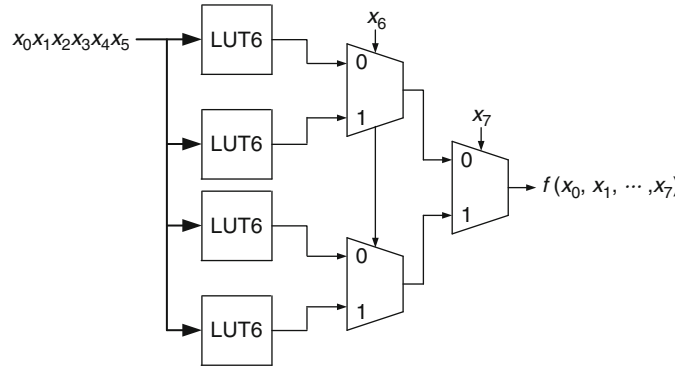


Fig. 2.58 Implementation of an 8-variable switching function

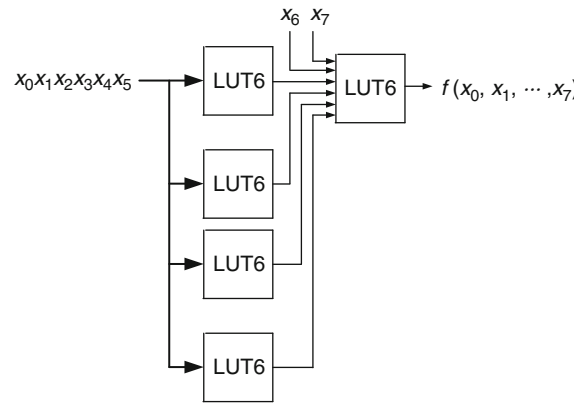


Fig. 2.59 Alternative circuit

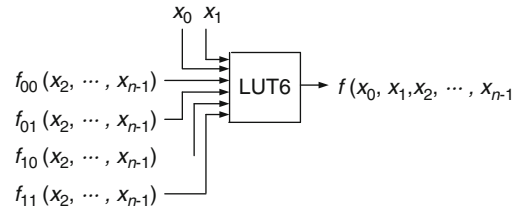


Fig. 2.60 Extraction of variables x_0 and x_1

$$f(x_0, x_1, \ \ldots, x_{n-1}) = \ \overline{x_0} \cdot \overline{x_1} \cdot f_{00}(x_2, \ldots, x_{n-1}) + \overline{x_0} \cdot x_1 \cdot f_{01}(x_2, \ \ldots, x_{n-1}) \\ + \ x_0 \cdot \overline{x_1} \cdot f_{10}(x_2, \ldots, x_{n-1}) + \ x_0 \cdot \ x_1 \cdot f_{11}(x_2, \ \ldots, x_{n-1}). \tag{2.42}$$

where

$$f_{00}(x_2, \ldots, x_{n-1}) = f(0, \ 0, x_2, \ \ldots, x_{n-1}), f_{01}(x_2, \ldots, x_{n-1}) = f(0, \ 1, x_2, \ \ldots, x_{n-1}), \\ f_{10}(x_2, \ldots, x_{n-1}) = f(1, \ 0, x_2, \ \ldots, x_{n-1}), f_{11}(x_2, \ldots, x_{n-1}) = f(1, \ 1, x_2, \ \ldots, x_{n-1})$$

are functions of $n - 2$ variables. The corresponding variable extraction circuit (Fig. 2.60) is a LUT6 that implements a function of six variables x_0, x_1, f_{00}, f_{01}, f_{10}, and f_{11} equal to

$$\overline{x_0} \cdot \overline{x_1} \cdot f_{00} + \overline{x_0} \cdot \ x_1 \cdot f_{01} + \ x_0 \cdot \ \overline{x_1} \cdot f_{10} + \ x_0 \cdot \ x_1 \cdot f_{11}.$$

Then a similar variable extraction can be performed with functions f_{00}, f_{01}, f_{10}, and f_{11} so that functions of $n - 4$ variables are obtained, and so on.

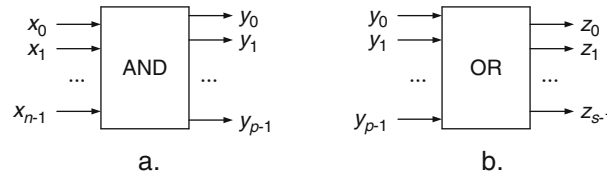


Fig. 2.61 AND plane and OR plane

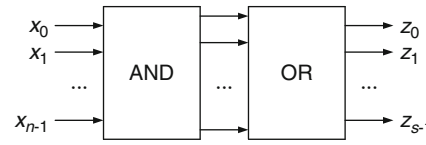
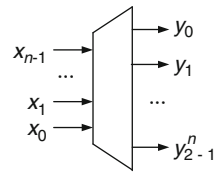
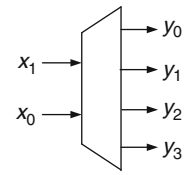


Fig. 2.62 Switching function implementation with two planes





x_1	x_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Fig. 2.63 Address decoders

2.7.3 Planes

Sometimes AND planes and OR planes are used to implement switching functions. An (n, p) AND plane (Fig. 2.61a) implements p functions y_j of n variables, where y_j is a product of literals (variable or inverse of a variable):

$$y_j = w_{j,0} w_{j,1} \ldots w_{j,n-1} \quad \text{where } w_{j,i} \in \{1, x_i, \overline{x}_i\}.$$

An (p, s) OR plane (Fig. 2.61b) implements s functions z_j of p variables, where z_j is a Boolean sum of variables:

$$z_j = w_{j,0} + w_{j,1} + \ldots + w_{j,p-1} \quad \text{where } w_{j,i} \in \{0, y_i\}.$$

Those planes can be configured when the corresponding integrated circuit (IC) is manufactured, or can be programmed by the user in which case they are called field programmable devices.

Any set of s switching functions that are expressed as Boolean sums of at most p products of at most n literals can be implemented by a circuit made up of an (n, p) AND plane and an (p, s) OR plane (Fig. 2.62): the AND plane generates p products of at most n literals and the OR plane generates s sums of at most p terms.

Depending on the manufacturing technology and on the manufacturer those AND-OR plane circuits receive different names such as programmable array of logic (PAL), Programmable Logic Array (PLA), Programmable Logic Device (PLD), and others.

2.7.4 Address Decoder and Tristate Buffers

Another type of useful component is the address decoder. An n-to-2^n address decoder (Fig. 2.63a) has n inputs and 2^n outputs and its function is defined as follows: if $x_{n-1}x_{n-2} \ldots x_0$ is the binary

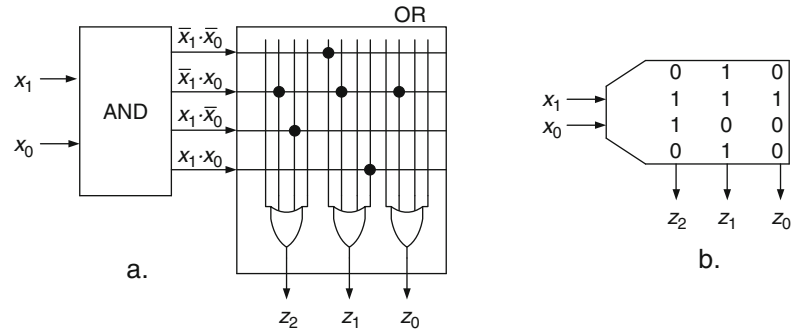


Fig. 2.64 AND-OR plane implementation of a ROM

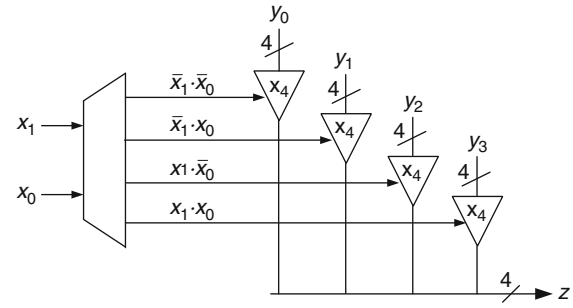


Fig. 2.65 4-Bit MUX4-1 implemented with an address decoder and four tristate buffers

representation of natural i, then $y_i = 1$ and all other outputs $y_j = 0$. As an example, a 2-to-4 address decoder and its truth table are shown in Fig. 2.63b.

In fact, an n-to-2^n address decoder implements the same function as an $(n, 2^n)$ AND plane that generates all n-variable minterms:

$$m_j = w_{j,0} w_{j,1} \ldots w_{j,n-1} \quad \text{where} \quad w_{j,i} \in \{x_i, \overline{x}_i\}.$$

By connecting an n-to-2^n address decoder to an $(2^n, s)$ OR plane, the obtained circuit implements the same function as a ROM storing $2^n s$-bit words. An example is given in Fig. 2.64a: the AND plane synthesizes the functions of a 2-to-4 address decoder and the complete circuit implements the same functions as the ROM of Fig. 2.64b.

The other common application of address decoders is the control of data buses. An example is given in Fig. 2.65: a 2-to-4 address decoder generates four signals that control four 4-bit tristate buffers. This circuit permits to connect a 4-bit output z to one among four 4-bit inputs y_0, y_1, y_2, or y_3 under the control of two address bits x_1 and x_0. Actually, the circuit of Fig. 2.65 realizes the same function as a 4-bit MUX4-1.

In Fig. 2.66a the circuit of Fig. 2.65 is used to connect one among four data sources (circuits A, B, C, and D) to a data destination (circuit E) under the control of two address bits. It executes the following algorithm:

```
case x1 x0 is
  when 00 => circuit_E = circuit_A;
  when 01 => circuit_E = circuit_B;
  when 00 => circuit_E = circuit_C;
  when 00 => circuit_E = circuit_D;
end case;
```

The usual symbol of this bus is shown in Fig. 2.66b.

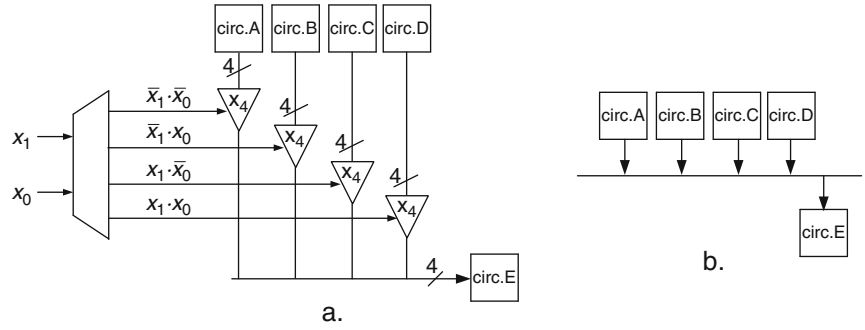


Fig. 2.66 A 4-bit data bus with four data sources

2.8 Programming Language Structures

The specification of digital systems by means of algorithms (Sect. 1.2.1) is a central aspect of this course. In this section the relation between some programming language instructions and digital circuits is analyzed. This relation justifies the use of hardware description languages (HDL) similar to programming languages, as well as the generation of synthesis tools able to translate HDL descriptions to circuits.

2.8.1 If Then Else

A first example of instruction that can be translated to a circuit is the conditional branch:

```
if a_condition then some_actions else other_actions;
```

As an example, consider the following binary decision algorithm. It computes the value of a switching function f of six variables x_0, x_1, y_0, y_1, y_2, and y_3.

Algorithm 2.4

```
if x1 = 0 then
  if x0 = 0 then f = y0; else f = y1; end if;
else
  if x0 = 0 then f = y2; else f = y3; end if;
end if;
```

This function can be implemented by the circuit of Fig. 2.67 in which the external conditional branch is implemented by the rightmost MUX2-1 and the two internal conditional branches are implemented by the leftmost MUX2-1s.

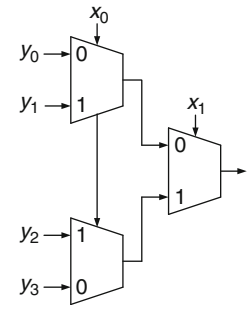


Fig. 2.67 Binary decision algorithm implementation

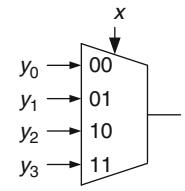


Fig. 2.68 Case instruction implementation

2.8.2 Case

A second example of instruction that can be translated to a circuit is the conditional switch:

```
case variable_identifier is
  when variable_value1 => actions1;
  when variable_value2 => actions2;
  ...
end case;
```

As an example, the preceding binary decision algorithm (Algorithm 2.4) is equivalent to the following, assuming that x has been previously defined as a 2-bit vector (x_0, x_1).

Algorithm 2.5

```
case x is
  when 00 => f = y0;
  when 01 => f = y1;
  when 10 => f = y2;
  when 11 => f = y3;
end case;
```

Function f can be implemented by a MUX4-1 (Fig. 2.68).

2.8.3 Loops

For-loops are a third example of easily translatable construct:

```
for variable_identifier in variable_range loop
  operations using the variable value
end loop;
```

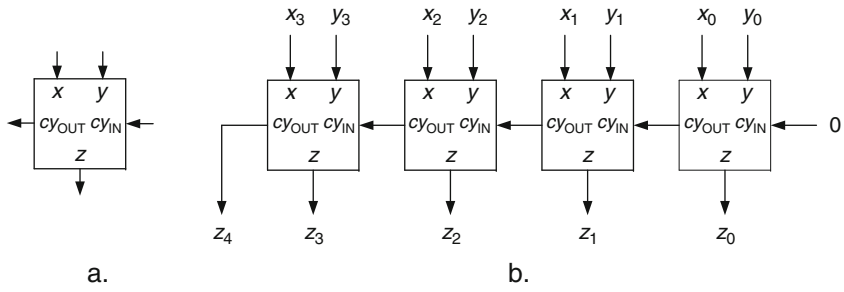


Fig. 2.69 4-Digit decimal adder

To this type of instruction can often be associated an iterative circuit. As an example, consider the following addition algorithm that computes $z = x + y$ where x and y are 4-digit decimal numbers, so that z is a 5-digit decimal number.

Algorithm 2.6 Addition of Two 4-Digit Naturals

```
cy0= 0;
for i in 0 to 3 loop
    --------- loop body:
    si= xi+ yi+ cyi;
    if si> 9 then zi= si- 10; cyi+1= 1;
    else zi= si; cyi+1= 0;
    end if;
    --------- end of loop body:
end loop;
z4= cy4;
```

The corresponding circuit is shown in Fig. 2.69b. It is an iterative circuit that consists of four identical blocks. Each of them is a 1-digit adder (Fig. 2.69a) that implements the loop body of Algorithm 2.6.

Comments 2.5

- Other (not combinational but sequential) loop implementation methods will be studied in Chap. 4.
- Not any loop can be implemented by means of an iterative combinational circuit. Consider a while-loop:

```
while a_condition loop operations end loop;
```

The loop body is executed as long as some condition (that can be modified by the operations) is true. If the maximum number of times that the condition will be true is either unknown or is a too large number, a sequential implementation (Chap. 4) must be considered.

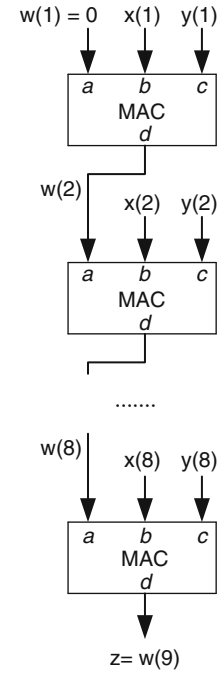


Fig. 2.70 Implementation of procedure calls

2.8.4 Procedure Calls

Procedure (or function) calls constitute a fundamental aspect of well-structured programs and can be associated to hierarchical circuit descriptions.

The following algorithm computes $z = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_8 \cdot y_8$. For that it makes several calls to a previously defined procedure multiply and accumulate (MAC) to which it passes four parameters a, b, c, and d. The procedure call MAC(a, b, c, d) executes $d = a + b \cdot c$.

Algorithm 2.7 $z = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_8 \cdot y_8$

```
w(1) = 0;
for i in 1 to 8 loop
  MAC(w(i), x(i), y(i), w(i+1));
end loop;
z = w(9);
```

Thus

$$w_2 = 0 + x_1 \cdot y_1 = x_1 \cdot y_1, w_2 = x_1 \cdot y_1 + x_2 \cdot y_2, w_3 = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3, \ldots,$$
$$z = w_9 = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3 + \ldots + x_8 \cdot y_8.$$

The corresponding circuit is shown in Fig. 2.70. Algorithm 2.7 is a for-loop to which is associated an iterative circuit. The loop body is a procedure call to which corresponds a component MAC whose functional specification is $d = a + b \cdot c$. This is an example of top-down hierarchical description: an iterative circuit structure (the top level) whose components are defined by their function and afterwards must be implemented (the down level).

2.8.5 Conclusion

There are several programming language constructs that can easily be translated to circuits. This fact justifies the use of formal languages to specify digital circuits, either classical programming languages such as C/C++ or specific HDL such as VHDL or Verilog. In this course VHDL will be used (Appendix A). The relation between programming language instructions and circuits also explains why it has been possible to develop software packages able to synthesize circuits starting from functional descriptions in some formal language.

2.9 Exercises

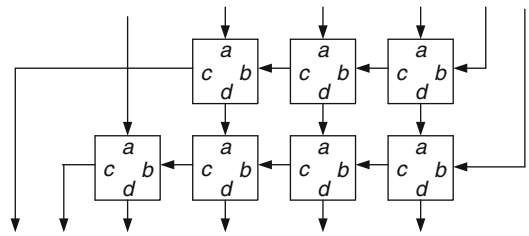
1. Synthesize with logic gates the function z of Table 2.3.
2. Generate Boolean expressions of functions f, g, and h of three variables x_2, x_1, and x_0 defined by the following table:

$x_2x_1x_0$	f	g	h
000	1	0	1
001	–	–	1
010	0	0	–
011	1	1	–
100	–	1	0
101	1	1	0
110	0	–	1
111	0	0	–

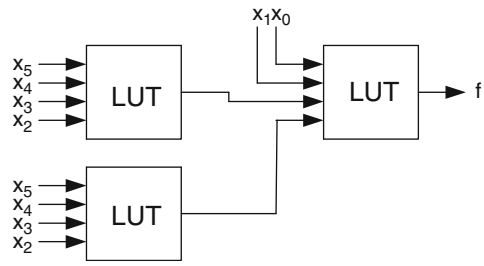
3. Simplify the following sets of cubes ($n = 4$):

   {0000, 0010, 01x1, 0110, 1000. 1010},
   {0001, 0011, 0100, 0101, 1100, 1110, 1011, 1010},
   {0000, 0010, 1000, 1010, 0101, 1101, 1111}.

4. The following circuit consists of seven identical components with two inputs a and b, and two outputs c and d. The maximum propagation time from inputs a or b to outputs c or d is equal to 0.5 ns. What is the maximum propagation time from any input to any output (in ns)?



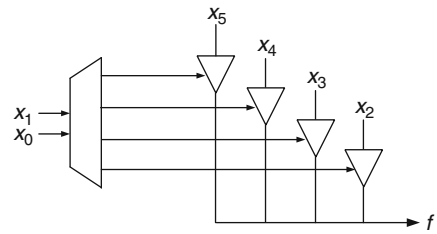
5. Compute an upper bound N_{max} and a lower bound N_{min} of the number N of functions that can be implemented by the following circuit.



6. Implement with MUX2-1 components the switching functions of three variables x_2, x_1, and x_0 defined by the following set of cubes:

   $\{11x,\ 101,\ 011\}$,
   $\{111,\ 100,\ 010,\ 001\}$,
   $\{1x1,\ 0x1\}$.

7. What set of cubes defines the function $f(x_5, x_4, x_3, x_2, x_1, x_0)$ implemented by the following circuit?



8. Minimize the following Boolean expression:

$$f(a,b,c,d) = a.b.c.d + \overline{a}.\overline{b} + a.b.\overline{c} + a.\overline{b} + a.\overline{c}.$$

9. Implement the circuits of Figs. 2.14, 2.16, and 2.17 with NAND gates.
10. Implement (2.34) with NAND gates.

References

Burch C (2005) Logisim. http://www.cburch.com/logisim/es/index.html
Karnaugh M (1953) The map method for synthesis of combinational logic circuits. Trans Inst Electr Eng (AIEE) Part I 72(9):593–599

Arithmetic Blocks

3

Arithmetic circuits are an essential part of many digital circuits and thus deserve a particular treatment. In this chapter implementations of the basic arithmetic operations are presented. Only operations with naturals (nonnegative integers) are considered. A much more detailed and complete presentation of arithmetic circuits can be found in Parhami (2000), Ercegovac and Lang (2004), Deschamps et al. (2006), and Deschamps et al. (2012).

3.1 Binary Adder

Binary adders have already been described several times (e.g., Figs. 2.5 and 2.27). Given two n-bit naturals x and y and an incoming carry bit cy_0, an n-bit adder computes an $(n + 1)$-bit number $s = x + y + cy_0$, in which s_n can be used as an outgoing carry bit cy_n. The classical pencil and paper algorithm, adapted to the binary system, is the following:

Algorithm 3.1 Binary Adder: $s = x + y + cy_0$

```
for i in 0 to n-1 loop
   s_i = x_i xor y_i xor cy_i;
   cy_i+1 = (x_i and y_i)) or (x_i and cy_i) or (y_i and cy_i);
end loop;
s_n = cy_n;
```

At each step

$$s_i = (x_i + y_i + cy_i) \bmod 2 = x_i \oplus y_i \oplus cy_i, \tag{3.1}$$

and $cy_{i+1} = 1$ if, and only if, at least two bits among x_i, y_i, and cy_i are equal to 1, a condition that can be expressed as follows:

$$s_{i+1} = x_i \cdot y_i \ + \ x_i \cdot cy_i \ + \ y_i \cdot cy_i. \tag{3.2}$$

The circuit that implements Algorithm 3.1 (a for-loop) is shown in Fig. 3.1. It consists of n identical blocks called full adders (FA) that implement the loop body of Algorithm 3.1 (3.1 and 3.2).

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_3

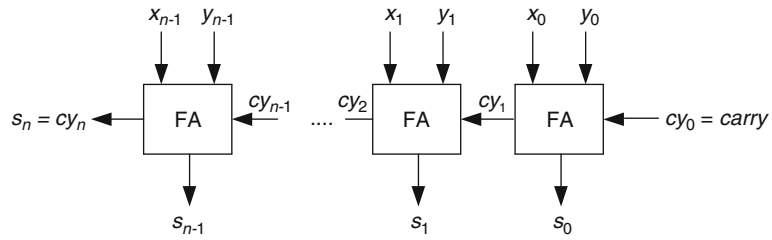


Fig. 3.1 n-Bit adder

3.2 Binary Subtractor

Given two n-bit naturals x and y and an incoming borrow bit b_0, an n-bit subtractor computes $d = x - y - b_0$. Thus $d \geq 0 - (2^n - 1) - 1 = -2^n$ and $d \leq (2^n - 1) - 0 - 0 = 2^n - 1$, so that d is a signed integer belonging to the range

$$-2^n \leq d \leq 2^n - 1.$$

The classical pencil and paper algorithm, adapted to the binary system, is used. At each step the difference $x_i - y_i - b_i$ is computed and expressed under the form

$$x_i - y_i - b_i = d_i - 2 \cdot b_{i+1} \text{ where } d_i \text{ and } b_{i+1} \in \{0, 1\}. \tag{3.3}$$

If $x_i - y_i - b_i < 0$ then $d_i = x_i - y_i - b_i + 2$ and $b_{i+1} = 1$; if $x_i - y_i - b_i \geq 0$ then $d_i = x_i - y_i - b_i$ and $b_{i+1} = 0$. At the end of step n the result is obtained under the form

$$d = -d_n \cdot 2^n + d_{n-1} \cdot 2^{n-1} + d_{n-2} \cdot 2^{n-2} + \ldots + d_0 \cdot 2^0 \tag{3.4}$$

where $d_n = b_n$ is the last borrow bit. This type of representation (3.4) in which the most significant bit d_n has a negative weight -2^n is the 2's complement representation of the signed integer d. In this representation d_n is the sign bit.

Example 3.1 Compute ($n = 4$) 0111 − 1001 − 1:

x_i:		0	1	1	1
y_i:		1	0	0	1
b_i:	1	0	0	1	1
d_i:	1	1	1	0	1

Conclusion: 0111 − 1001 − 1 = 11101. In decimal: 7 − 9 − 1 = −16 + 13 = −3.
By reducing both members of (3.3) modulo 2 the following relation is obtained:

$$d_i = x_i \oplus y_i \oplus b_i. \tag{3.5}$$

On the other hand $b_{i+1} = 1$ if, and only if, $x_i - y_i - b_i < 0$, that is, when $x_i = 0$ and either y_i or b_i is equal to 1, or when both y_i and b_i are equal to 1. This condition can be expressed as follows:

$$b_{i+1} = \overline{x_i} \cdot y_i + \overline{x_i} \cdot b_i + y_i \cdot b_i. \tag{3.6}$$

The following algorithm computes d.

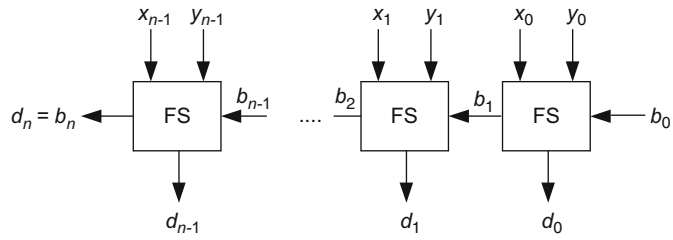


Fig. 3.2 *n*-Bit subtractor

Algorithm 3.2 Binary Subtractor: $d = x - y - b_0$

```
for i in 0 to n-1 loop
   d_i = x_i xor y_i xor b_i;
   b_{i+1} = (not(x_i) and y_i) or (not(x_i) and b_i)
      or (y_i and b_i);
end loop;
d_n = b_n;
```

The circuit that implements Algorithm 3.2 (a for-loop) is shown in Fig. 3.2. It consists of n identical blocks called full subtractors (FS) that implement the loop body of Algorithm 3.2 (3.5 and 3.6).

3.3 Binary Adder/Subtractor

Given two n-bit naturals x and y and a 1-bit control input a/s, an n-bit adder/subtractor computes $z = [x + y] \bmod 2^n$ if $a/s = 0$ and $z = [x - y] \bmod 2^n$ if $a/s = 1$. To compute z, define $\overline{y}$ as being the natural deduced from y by inverting all its bits: $\overline{y} = \overline{y_{n-1}}\ \overline{y_{n-2}} \ldots \overline{y_0}$, and check that

$$\overline{y} = \left(1 - y_{n-1}\right) \cdot 2^{n-1} + \left(1 - y_{n-2}\right) \cdot 2^{n-2} + \ldots + (1 - y_0) \cdot 2^0 = 2^n - 1 - y. \tag{3.7}$$

Thus z can be computed as follows:

$$z = [x + w + a/s] \bmod 2^n, \text{ where } w = y \text{ if } a/s = 0 \text{ and } w = \overline{y} \text{ if } a/s = 1. \tag{3.8}$$

In other words $w_i = a/s \oplus y_i, \forall\, i = 0$ to $n - 1$.

Algorithm 3.3 Binary Adder/Subtractor

```
for i in 0 to n-1 loop
   w_i = a/s xor y_i;
end loop;
z = (x + w + a/s) mod 2^n;
```

The circuit that implements Algorithm 3.3 is shown in Fig. 3.3. It consists of an n-bit adder and n XOR2 gates. An additional XOR2 gate computes *ovf* (overflow): if $a/s = 0$, $ovf = 1$ if, and only if, $cy_n = 1$ and thus $x + y \geq 2^n$; if $a/s = 1$, $ovf = 1$ if, and only if, $cy_n = 0$ and thus $x + (2^n - 1 - y) + 1 < 2^n$, that is to say if $x - y < 0$.

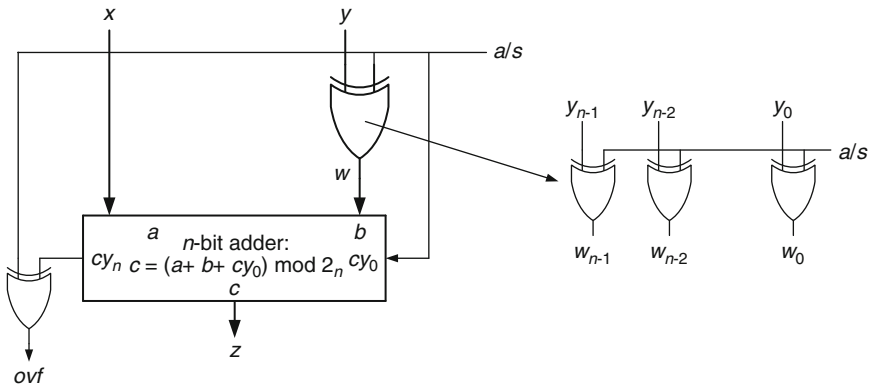


Fig. 3.3 *n*-Bit adder/subtractor

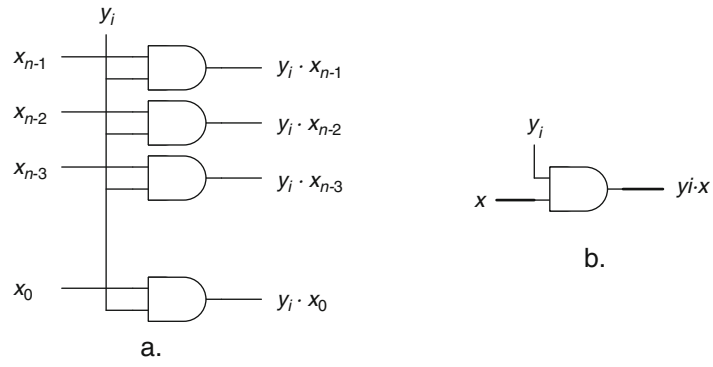


Fig. 3.4 Multiplication by y_i (circuit and symbol)

3.4 Binary Multiplier

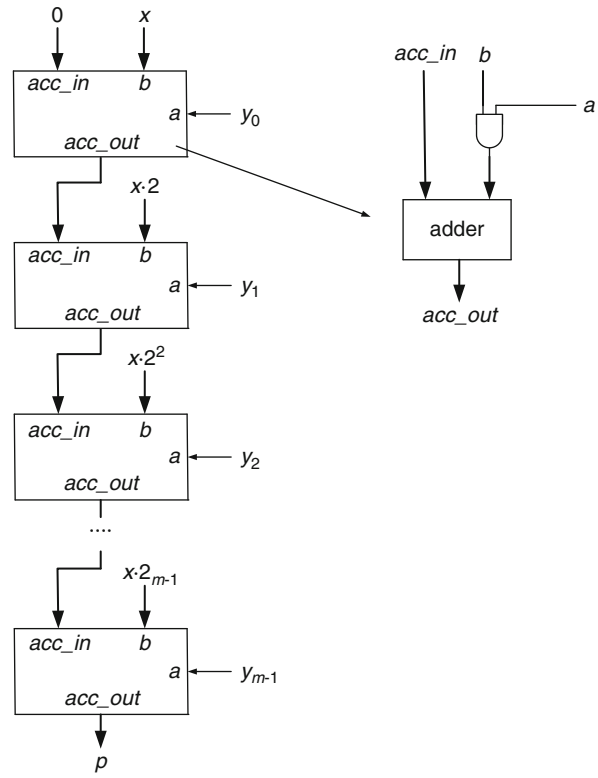
Given an n-bit natural x and an m-bit natural y a multiplier computes $p = x \cdot y$. The maximum value of p is $(2^n - 1) \cdot (2^m - 1) < 2^{n+m}$ so that p is an $(n + m)$-bit natural. If $y = y_{m-1} \cdot 2^{m-1} + y_{m-2} \cdot 2^{m-2} + \ldots + y_1 \cdot 2 + y_0$, then

$$p = x \cdot y_{m-1} \cdot 2^{m-1} + x \cdot y_{m-2} \cdot 2^{m-2} + \ldots + x \cdot y_1 \cdot 2 + x \cdot y_0. \tag{3.9}$$

The preceding expression can be computed as follows. First compute a set of partial products $p_0 = x \cdot y_0$, $p_1 = x \cdot y_1 \cdot 2$, $p_2 = x \cdot y_2 \cdot 2^2$, $\ldots$, $p_{m-1} = x \cdot y_{m-1} \cdot 2^{m-1}$, and then add the m partial products: $p = p_0 + p_1 + p_2 + \ldots + p_{m-1}$. The computation of each partial product $p_i = x \cdot y_i \cdot 2^i$ is very easy. The product $x \cdot y_i$ is computed by a set of AND2 gates (Fig. 3.4) and the multiplication by 2^i amounts to adding i 0s to the right of the binary representation of $x \cdot y_i$. For example, if $i = 5$ and $x \cdot y_5 = 10010110$ then $x \cdot y_5 \cdot 2^5 = 1001011000000$.

The computation of $p = p_0 + p_1 + p_2 + \ldots + p_{m-1}$ can be executed by a sequence of 2-operand additions: $p = (\ldots(((0 + p_0) + p_1) + p_2)\ldots) + p_{m-1}$. The following algorithm computes p.

Fig. 3.5 Binary multiplier



Algorithm 3.4 Binary Multiplier: $p = x \cdot y$ (Right to Left Algorithm)

```
acc₀ = 0;
for i in 0 to m-1 loop
   acc_{i+1} = acc_i + x·y_i·2^i;
end loop;
p = acc_m;
```

Example 3.2 Compute ($n = 5, m = 4$) $11101 \cdot 1011$ (in decimal $29 \cdot 11$). The values of acc_i are the following:

acc_0:	0	0	0	0	0	0	0	0	0
acc_1:	0	0	0	0	1	1	1	0	1
acc_2:	0	0	1	0	1	0	1	1	1
acc_3:	0	0	1	0	1	0	1	1	1
acc_4:	1	0	0	1	1	1	1	1	1

Result: $p = 100111111$ (in decimal 319).

The circuit that implements Algorithm 3.4 (a for-loop) is shown in Fig. 3.5. It consists of m identical blocks that implement the loop body of Algorithm 3.4:

$$acc_{\text{out}} = acc_{\text{in}} + b \cdot a \quad \text{where} \quad b = x \cdot 2^i \quad \text{and} \quad a = y_i.$$

Comment 3.1

The building block of Fig. 3.5 computes $acc_{\text{in}} + b \cdot a$. At step i input $b = x \cdot 2^i$ is an $(n + i)$-bit number. In particular at step $m - 1$ it is an $(n + m - 1)$-bit number. Thus, if all blocks are identical,

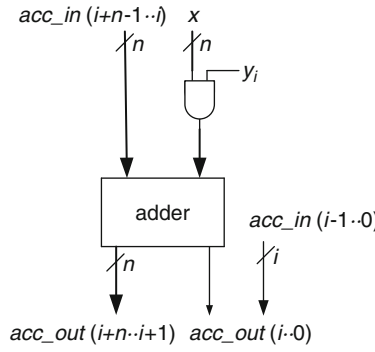


Fig. 3.6 Optimized block

they must include an $(n + m - 1)$-bit adder and $n + m - 1$ AND2 gates. Nevertheless, for each i this building block can be optimized. At step number i it computes $acc_{\text{out}} = acc_{\text{in}} + x \cdot y_i \cdot 2^i$ where acc_{in} is an $(i + n)$-bit number (at each step one bit is added). On the other hand the rightmost i bits of $x \cdot y_i \cdot 2^i$ are equal to 0. Thus

$$acc_{\text{out}}(i + n \cdot \cdot i) = acc_{\text{in}}(i + n - 1 \cdot \cdot i) + x \cdot y_i,$$

$$acc_{\text{out}}(i - 1 \cdot \cdot 0) = acc_{\text{in}}(i - 1 \cdot \cdot 0).$$

The corresponding optimized block is shown in Fig. 3.6. Each block contains an n-bit adder and n AND2 gates.

3.5 Binary Divider

Division is the more complex operation. Given two naturals x and y their quotient $q = x/y$ is usually not an integer. It is a so-called rational number. In many cases it is not even a fixed-point number. The desired accuracy must be taken into account. The quotient q with an accuracy of p fractional bits is defined by the following relation:

$$x/y = q + e \text{ where } q \text{ is a multiple of } 2^{-p} \text{ and } e < 2^{-p}. \tag{3.10}$$

In other words q is a fixed-point number with p fractional bits

$$q = q_{m-1}q_{m-2} \cdots q_0 \cdot q_{-1}q_{-2} \cdots q_{-p} \tag{3.11}$$

such that the error $e = x/y - q$ is smaller than 2^{-p}.

Most division algorithms work with naturals x and y such that $x < y$, so that

$$q = 0 \,.q_{-1}q_{-2} \cdots q_{-p}. \tag{3.12}$$

Consider the following sequence of integer divisions by y with $x = r_0 < y$:

$$\begin{aligned}
2 \cdot r_0 &= q_{-1} \cdot y + r_1 \text{ with } r_1 < y, \\
2 \cdot r_1 &= q_{-2} \cdot y + r_2 \text{ with } r_2 < y, \\
&\cdots \\
2 \cdot r_{p-2} &= q_{-p+1} \cdot y + r_{p-1} \text{ with } r_{p-1} < y, \\
2 \cdot r_{p-1} &= q_{-p} \cdot y + r_p \text{ with } r_p < y.
\end{aligned} \tag{3.13}$$

At each step q_{-i} and r_i are computed in function of r_{i-1} and y so that the following relation holds true:

$$2 \cdot r_{i-1} = q_{-i} \cdot y + r_i. \tag{3.14}$$

For that

- Compute $d = 2 \cdot r_{i-1} - y$.
- If $d < 0$ then $q_{-i} = 0$ and $r_i = 2 \cdot r_{i-1}$; else $q_{-i} = 1$ and $r_i = d$.

Property 3.1

$$x/y = 0.q_{-1}q_{-2}\ldots q_{-p+1}q_{-p} + \left(r_p/y\right) \cdot 2^{-p} \text{ with } \left(r_p/y\right) \cdot 2^{-p} < 2^{-p}.$$

Proof Multiply the first equation of (3.13) by 2^{p-1}, the second by 2^{p-2}, and so on. Thus

$$\begin{aligned}
2^p \cdot r_0 &= 2^{p-1} \cdot q_{-1} \cdot y + 2^{p-1} \cdot r_1 \text{ with } r_1 < y,\\
2^{p-1} \cdot r_1 &= 2^{p-2} \cdot q_{-2} \cdot y + 2^{p-2} \cdot r_2 \text{ with } r_2 < y,\\
&\ldots\\
2^2 \cdot r_{p-2} &= 2 \cdot q_{-p+1} \cdot y + 2 \cdot r_{p-1} \text{ with } r_{p-1} < y,\\
2 \cdot r_{p-1} &= q_{-p} \cdot y + r_p \text{ with } r_p < y.
\end{aligned}$$

Then add up the p equations:

$$2^p \cdot r_0 = 2^{p-1} \cdot q_{-1} \cdot y + 2^{p-2} \cdot q_{-2} \cdot y + \ldots + 2 \cdot q_{-p+1} \cdot y + q_{-p} \cdot y + r_p \text{ with } r_p < y,$$

so that

$$x = \left(2^{-1} \cdot q_{-1} + 2^{-2} \cdot q_{-2} + \ldots + 2^{-p+1} \cdot q_{-p+1} + 2^{-p} \cdot q_{-p}\right) \cdot y + r_p \cdot 2^{-p} \text{ with } r_p \cdot 2^{-p} < y \cdot 2^{-p},$$

and

$$x/y = 0.q_{-1}q_{-2}\ldots q_{-p+1}q_{-p} + \left(r_p/y\right) \cdot 2^{-p} \text{ with } \left(r_p/y\right) \cdot 2^{-p} < 2^{-p}.$$

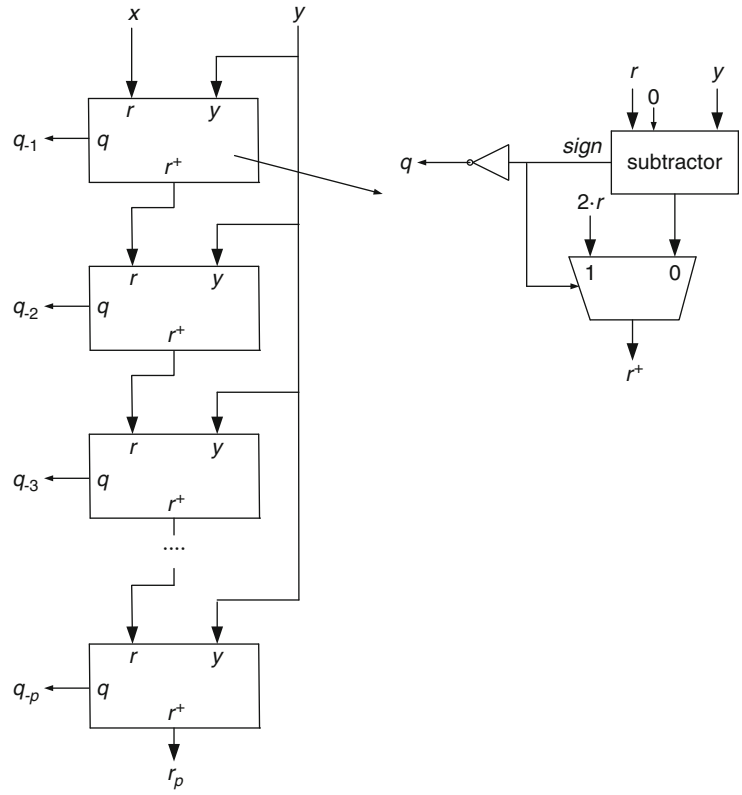
Example 3.3 Compute 21/35 with an accuracy of 6 bits:

$2 \cdot 21 = 1 \cdot 35 + 7$
$2 \cdot 7 = 0 \cdot 35 + 14$
$2 \cdot 14 = 0 \cdot 35 + 28$
$2 \cdot 28 = 1 \cdot 35 + 21$
$2 \cdot 21 = 1 \cdot 35 + 7$
$2 \cdot 7 = 0 \cdot 35 + 14$

Thus $q = 0.100110$. In decimal: $q = 38/2^6$. Error $= 21/35 - 38/2^6 = 0.6 - 0.59375 = 0.00625 < 2^{-6} = 0.015625$.

The following algorithm computes q with an accuracy of p fractional bits.

Fig. 3.7 Binary divider



Algorithm 3.5 Binary Divider: $q \cong x/y$, Error $< 2^{-p}$ (Restoring Algorithm)

```
r0 = x;
for i in 1 to p loop
   d = 2·ri-1 - y;
   if d < 0 then q-i = 0; ri = 2·ri-1;
   else q-i = 1; ri = d;
   end if;
end loop;
```

The circuit that implements Algorithm 3.5 (a for-loop) is shown in Fig. 3.7. It consists of p identical blocks that implement the loop body of Algorithm 3.5.

3.6 Exercises

1. An integer x can be represented under the form $(-1)^s \cdot m$ where s is the sign of x and m is its magnitude (absolute value). Design an n-bit sign-magnitude adder/subtractor.
2. An incrementer-decrementer is a circuit with two n-bit inputs x and m, one binary control input *up/down*, and one n-bit output z. If *up/down* $= 0$, it computes $z = (x + 1) \bmod m$, and if *up/down* $= 1$, it computes $z = (x - 1) \bmod m$. Design an n-bit incrementer-decrementer.
3. Consider the circuit of Fig. 3.5 with the optimized block of Fig. 3.6. The n-bit adder of Fig. 3.6 can be implemented with n 1-bit adders (full adders). Define a 1-bit multiplier as being a component

with four binary inputs a, b, c, d, and two binary outputs e and f, that computes $a \cdot b + c + d$ and expresses the result as $2 \cdot e + f$ (a 2-bit number). Design an n-bit-by-m-bit multiplier consisting of 1-bit multipliers.

4. Synthesize a $2n$-bit-by-$2n$-bit multiplier using n-bit-by-n-bit multipliers and n-bit adders as components.
5. A mod m reducer is a circuit with two n-bit inputs x and m ($m > 2$) and one n-bit output $z = x \bmod m$. Synthesize a mod m reducer.

References

Deschamps JP, Gioul G, Sutter G (2006) Synthesis of arithmetic circuits. Wiley, New York
Deschamps JP, Sutter G, Cantó E (2012) Guide to FPGA implementation of arithmetic functions. Springer, Netherlands
Ercegovac M, Lang T (2004) Digital arithmetic. Morgan Kaufmann Publishers, San Francisco
Parhami B (2000) Computer arithmetic. Oxford University Press, Oxford

Sequential Circuits

4

The digital systems that have been defined and implemented in the preceding chapters are combinational circuits. If the component delays are not taken into account, that means that the value of their output signals only depends on the values of their input signals at the same time. However, many digital system specifications cannot be implemented by combinational circuits because the value of an output signal could be a function of not only the value of the input signals at the same time, but also the value of the input signals at preceding times.

4.1 Introductory Example

Consider the vehicle access control system of Fig. 4.1. It consists of

- A gate that can be raised and lowered by a motor
- A push button to request the access
- Two sensors that detect two particular gate positions (upper and lower)
- A sensor that detects the presence of a vehicle within the gate area

The motor control system has four binary input signals:

- *Request* equal to 1 when there is an entrance request (push button)
- *Lower* equal to 1 when the gate has been completely lowered
- *Upper* equal to 1 when the gate has been completely raised
- *Vehicle* equal to 1 if there is a vehicle within the gate area

The binary output signals *on*/*off* and *up*/*down* control the motor:

- To raise the gate *on*/*off* $= 1$ and *up*/*down* $= 1$
- To lower the gate *on*/*off* $= 1$ and *up*/*down* $= 0$
- To maintain the gate open or closed *on*/*off* $= 0$

The motor control system cannot be implemented by a combinational circuit. As an example, if at some time *request* $= 0$, *vehicle* $= 0$, *upper* $= 0$, and *lower* $= 0$, this set of input signal values could

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_4

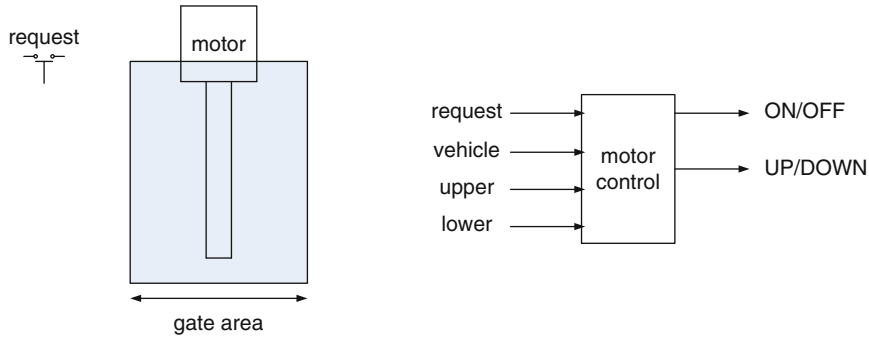


Fig. 4.1 Vehicle access control

correspond to two different situations: (1) a vehicle is present in front of the gate, the request button has been pushed and released, and the gate is moving up, or (2) a vehicle has got in and the gate is moving down. In the first case *on*/*off* = 1 and *up*/*down* = 1; in the second case *on*/*off* = 1 and *up*/*down* = 0.

In conclusion, the values of the signals that control the motor depend on the following sequence of events:

1. Wait for *request* = 1 (entrance request)
2. Raise the gate
3. Wait for *upper* = 1 (gate completely open)
4. Wait for *vehicle* = 0 (gate area cleared)
5. Lower the gate
6. Wait for *lower* = 1 (gate completely closed)

A new entrance request is not attended until this sequence of events is completed.

Conclusion: Some type of memory is necessary in order to store the current step number (1–6) within the sequence of events.

4.2 Definition

Sequential circuits are digital systems with memory. They implement systems whose output signal values depend on the input signal values at times t (the current time), $t - 1$, $t - 2$, and so on (the precise meaning of $t - 1$, $t - 2$, etc. will be defined later). Two simple examples are sequence detectors and sequence generators.

Example 4.1 (Sequence Detector) Implement a circuit (Fig. 4.2a) with a decimal input x and a binary output y. It generates an output value $y = 1$ every time that the four latest inputted values were 1 5 5 7. It is described by the following instruction in which t stands for the current time:

```
if x(t-3) = 1 AND x(t-2) = 5 AND x(t-2) = 5
AND x(t) = 7 then y = 1; else y = 0; end if;
```

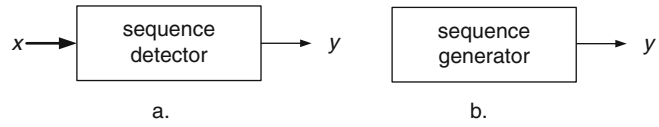


Fig. 4.2 Sequence detector and sequence generator

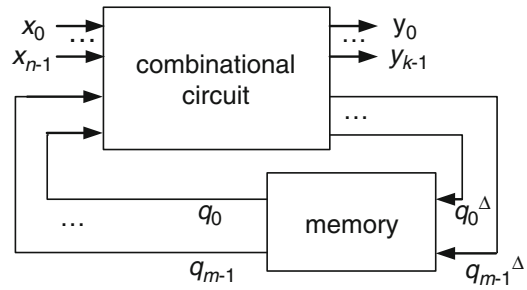


Fig. 4.3 Sequential circuit

Thus, the corresponding circuit must store $x(t-3)$, $x(t-2)$, and $x(t-1)$ and generates y in function of the stored values and of the current value of x.

Example 4.2 (Sequence Generator) Implement a circuit (Fig. 4.2b) with a binary output y that continuously generates the output sequence $011011011011\cdots$. It is described by the following instruction in which t stands for the current time:

```
if y(t-2) = 1 AND y(t-1) = 1 then y = 0; else y = 1; end if;
```

The corresponding circuit must store $y(t-2)$ and $y(t-1)$ and generates the current value of y in function of the stored values. Initially ($t = 0$) the stored values $y(-2)$ and $y(-1)$ are equal to 1 so that the first output value of y is 0.

The general structure of a sequential circuit is shown in Fig. 4.3. It consists of

- A combinational circuit that implements $k + m$ switching functions $y_0, y_1, \ldots, y_{k-1}, {q_0}^{\Delta}, {q_1}^{\Delta}, \ldots, {q_{m-1}}^{\Delta}$ of $n + m$ variables $x_0, x_1, \ldots, x_{n-1}, q_0, q_1, \ldots, q_{m-1}$
- A memory that stores an m-bit vector

The combinational circuit inputs $x_0, x_1, \ldots, x_{n-1}$ are inputs of the sequential circuit while $(q_0, q_1, \ldots, q_{m-1})$ is an m-bit vector read from the memory. The combinational circuit outputs $y_0, y_1, \ldots, y_{k-1}$ are outputs of the sequential circuit while $({q_0}^{\Delta}, {q_1}^{\Delta}, \ldots, {q_{m-1}}^{\Delta})$ is an m-bit vector written to the memory. The way the memory is implemented and the moments when the memory contents $(q_0, q_1, \ldots, q_{m-1})$ are updated and replaced by $({q_0}^{\Delta}, {q_1}^{\Delta}, \ldots, {q_{m-1}}^{\Delta})$ will be defined later.

With this structure, the output signals $y_0, y_1, \ldots, y_{k-1}$ depend not only on the current value of the input signals $x_0, x_1, \ldots, x_{n-1}$ but also on the memory contents $q_0, q_1, \ldots, q_{m-1}$. The values of $q_0, q_1, \ldots, q_{m-1}$ are updated at time $\ldots\ t-1, t, t+1, \ldots$ with new values ${q_0}^{\Delta}, {q_1}^{\Delta}, \ldots, {q_{m-1}}^{\Delta}$ that are generated by the combinational circuit.

The following terminology is commonly used:

- $x_0, x_1, \ldots, x_{n-1}$ are the *external inputs*
- $y_0, y_1, \ldots, y_{k-1}$ are the *external outputs*

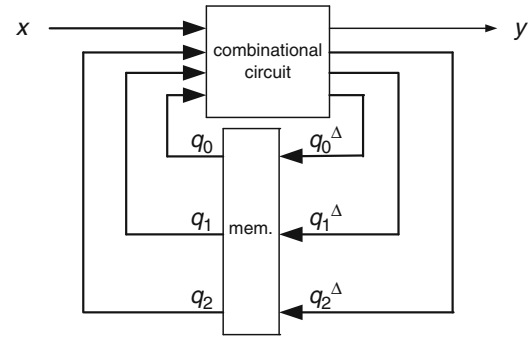


Fig. 4.4 Sequence detector implementation

- $(q_0, q_1, \ldots, q_{m-1})$ is the *internal state*
- $(q_0^{\Delta}, q_1^{\Delta}, \ldots, q_{m-1}^{\Delta})$ is the *next state*

To summarize,

- The memory stores the internal state.
- The combinational circuit computes the value of the external outputs and the next state in function of the external inputs and of the current internal state.
- The internal state is updated at every time unit $\ldots t-1, t, t+1, \ldots$ by replacing q_0 by q_0^{Δ}, q_1 by q_1^{Δ}, and so on.

Example 4.3 (Sequence Detector Implementation) The sequence detector of Example 4.1 can be implemented by the sequential circuit of Fig. 4.4 in which x, q_0, q_1, q_2, q_0^{Δ}, q_1^{Δ}, and q_2^{Δ} are 4-bit vectors that represent decimal digits. The memory must store the three previous values of x that are $q_0 = x(t-1)$, $q_1 = x(t-2)$, and $q_2 = x(t-3)$. For that

$$q_0^{\Delta} = x,\ q_1^{\Delta} = q_0, q_2^{\Delta} = q_1. \tag{4.1}$$

The output y is defined as follows:

$$y = 1 \text{ if, and only if, } q_2 = 1 \,\text{AND}\, q_1 = 5 \,\text{AND}\, q_0 = 5 \,\text{AND}\, x = 7. \tag{4.2}$$

Equations 4.1 and 4.2 define the combinational circuit function.

In the previous definitions and examples the concept of current time t is used but it has not been explicitly defined. To synchronize a sequential circuit and to give sense to the concept of current time and, in particular, to define the moments when the internal state is updated, a clock signal must be generated. It is a square wave signal (Fig. 4.5) with period T. The positive edges of this clock signal define the times that have been called $\ldots t-1, t, t+1, \ldots$ and are expressed in multiples of the clock signal period T. In particular the positive edges define the moments when the internal state is replaced by the next state. In Fig. 4.5 some commonly used terms are defined.

- Positive edge: a transition of the clock signal from 0 to 1
- Negative edge: a transition of the clock signal from 1 to 0
- Cycle: section of a clock signal that corresponds to one period
- Frequency: the number of cycles per second $(1/T)$
- Positive pulse: the part of a clock signal cycle where $clock = 1$
- Negative pulse: the part of a clock signal cycle where $clock = 0$

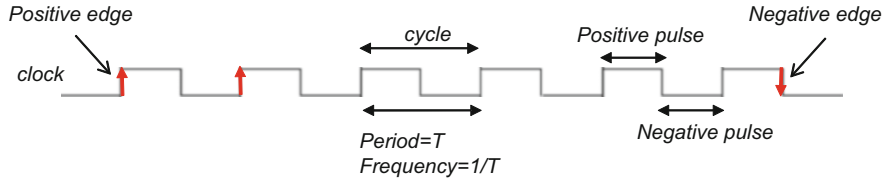


Fig. 4.5 Clock signal

Comment 4.1
Instead of using the positive edges of the clock signal to synchronize the circuit operations, the negative edges could be used. This is an essential part of the specification of a sequential circuit: positive edge triggered or negative edge triggered.

4.3 Explicit Functional Description

Explicit functional descriptions of combinational circuits (Sect. 1.2.1) are tables that define the output signal values associated to all possible combinations of input signal values. In the case of sequential circuits all possible internal states must also be considered: to different internal states correspond different relations between input and output signals.

4.3.1 State Transition Graph

A state transition graph consists of a set of vertices that correspond to the internal states and of a set of directed edges that define the internal state transitions and the output signal values in function of the input signal values.

Example 4.4 The graph of Fig. 4.6b defines a sequential circuit (Fig. 4.6a) that has three internal states A, B, and C encoded with two binary variables q_0 and q_1 that are stored in the memory block; it has a binary input signal x and a binary output signal y. It works as follows:

- If the internal state is A and if $x = 0$ then the next state is C and $y = 0$.
- If the internal state is A and if $x = 1$ then the next state is A and $y = 1$.
- If the internal state is B then (whatever x) the next state is A and $y = 0$.
- If the internal state is C and if $x = 0$ then the next state is C and $y = 0$.
- If the internal state is C and if $x = 1$ then the next state is B and $y = 1$.

To complete the combinational circuit (Fig. 4.6a) specification it remains to choose the encoding of states A, B, and C, for example:

$$A : q_0q_1 = 00, B : q_0q_1 = 01, C : q_0q_1 = 10. \tag{4.3}$$

The following case instruction defines the combinational circuit function:

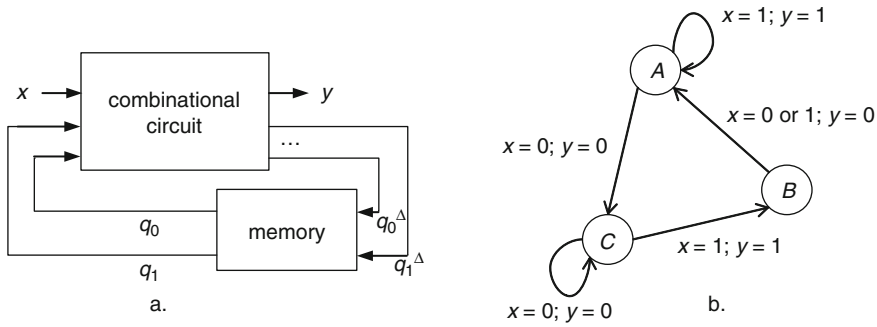


Fig. 4.6 Example of state transition graph (Mealy model)

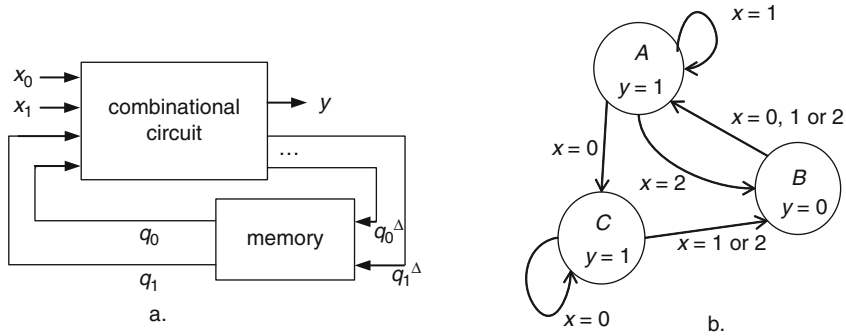


Fig. 4.7 Example of state transition graph (Moore model)

```
case q0q1 is
   when 00 => if x = 0 then q0Δq1Δ = 10; y = 0;
              else q0Δq1Δ = 00; y = 1; end if;
   when 01 => q0Δq1Δ = 00; y = 0;
   when 10 => if x = 0 then q0Δq1Δ = 10; y = 0;
              else q0Δq1Δ = 01; y = 1; end if;
   when others => q0Δq1Δ = don't care; y = don't care;
end case;
```

The clock signal (Sect. 4.2) is not represented in Fig. 4.6a but it is implicitly present and it is responsible for the periodic updating of the internal state.

The way that the external output signal values are defined in Fig. 4.6b corresponds to the so-called Mealy model: the value of y depends on the current internal state and on the current value of the input signal x. In the following example another method is used.

Example 4.5 The graph of Fig. 4.7b. defines a sequential circuit (Fig. 4.7a) that has three internal states A, B, and C encoded with two binary variables q_0 and q_1 that are stored in the memory block; it has two binary input signals x_0 and x_1 that encode a ternary digit $x \in \{0, 1, 2\}$ and a binary output signal y. It works as follows:

- If the internal state is A and if $x = 0$ then the next state is C and $y = 1$.
- If the internal state is A and if $x = 1$ then the next state is A and $y = 1$.

- If the internal state is A and if $x = 2$ then the next state is B and $y = 1$.
- If the internal state is B then (whatever x) the next state is A and $y = 0$.
- If the internal state is C and if $x = 0$ then the next state is C and $y = 1$.
- If the internal state is C and if $x = 1$ or 2 then the next state is B and $y = 1$.

To complete the combinational circuit (Fig. 4.7a) specification it remains to choose the encoding of states A, B, and C, for example the same as before (4.3). The following case instruction defines the combinational circuit function:

```
case q0q1 is
   when 00 => if x1x0 = 00 then q0Δq1Δ = 10;
              elsif x1x0 = 01 then q0Δq1Δ = 00;
              elsif x1x0 = 10 then q0Δq1Δ = 01;
              else q0Δq1Δ = don't care; end if;
              y = 1;
   when 01 => if x1x0 = 11 then q0Δq1Δ = don't care;
              else q0Δq1Δ = 00; end if;
              y = 0;
  when 10 => if x1x0 = 00 then q0Δq1Δ = 10;
              elsif (x1x0 = 01) or (x1x0 = 10)
              then q0Δq1Δ = 01;
              else q0Δq1Δ = don't care; end if;
              y = 1;
   when others => q0Δq1Δ = don't care; y = don't care;
end case;
```

In this case, the value of y only depends on the current internal state. It is the so-called Moore model: the value of y only depends on the current internal state; it does not depend on the current value of the input signals x_0 and x_1.

To summarize, two graphical description methods have been described. In both cases it is a graph whose vertices correspond to the internal states of the sequential circuit and whose directed edges are labelled with the input signal values that cause the transition from a state to another. They differ in the way that the external output signals are defined.

- In the first case (Example 4.4) the external output values are a function of the internal state and of the external input values. The directed edges are labelled with both the input signal values that cause the transition and with the corresponding output signal values. It is the Mealy model.
- In the second case (Example 4.5) the external output values are a function of the internal state. The directed edges are labelled with the input signal values that cause the transition and the vertices with the corresponding output signal values. It is the Moore model.

Observe that a Moore model is a particular case of Mealy model in which all edges whose origin is the same internal state are labelled with the same output signal values. As an example the graph of Fig. 4.8 describes the same sequential circuit as the graph of Fig. 4.7b. Conversely it can be demonstrated that a sequential circuit defined by a Mealy model can also be defined by a Moore model but, generally, with more internal states.

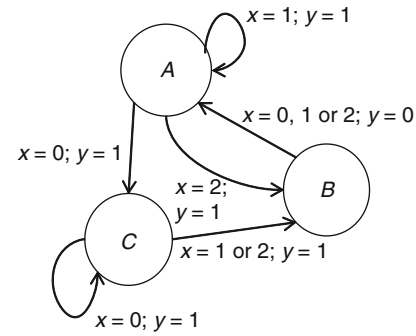


Fig. 4.8 Mealy model of Fig. 4.7b

Fig. 4.9 Photo of a robot vacuum cleaner (courtesy of iRobot Corporation)

4.3.2 Example of Explicit Description Generation

Given a functional specification of a sequential circuit, for example in a natural language, how can a state transition graph be defined? There is obviously no systematic and universal method to translate an informal specification to a state transition graph. It is mainly a matter of common sense and imagination. As an example, consider the circuit that controls a robot vacuum cleaner (the photo of a commercial robot is shown in Fig. 4.9).

To make the example more tractable a simplified version of the robot is defined:

- The robot includes a sensor that generates a binary signal $OB = 1$ when it detects an obstacle in front of it.
- The robot can execute three orders under the control of two binary inputs LR (left rotate) and RR (right rotate): move forward ($LR = RR = 0$), turn 90° to the left ($LR = 1, RR = 0$), and turn 90° to the right ($LR = 0, RR = 1$).

The specification of the robot control circuit is the following:

- If there is no obstacle: move forward.
- When an obstacle is detected: turn to the right until there is no more obstacle.
- The next time an obstacle is detected: turn to the left until there is no more obstacle.
- The next time an obstacle is detected: turn to the right until there is no more obstacle, and so on.

This behavior cannot be implemented by a combinational circuit. In order to take a decision it is not enough to know whether there is an obstacle or not; it is necessary to know the latest ordered movements:

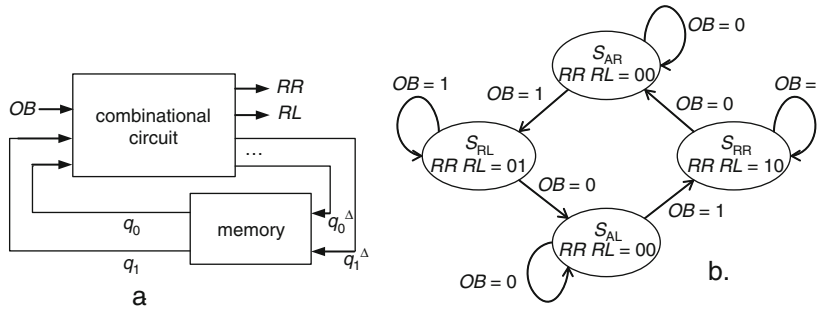


Fig. 4.10 Robot control circuit

- If the previous command was turn to the right and if there is no obstacle then move forward.
- If the previous command was turn to the right and there is still an obstacle then keep turning to the right.
- If the previous command was turn to the left and if there is no obstacle then move forward.
- If the previous command was turn to the left and there is still an obstacle then keep turning to the left.
- If the previous command was move forward and if there is no obstacle then keep moving forward.
- If the previous command was move forward and there is an obstacle and the latest rotation was to the left then turn to the right.
- If the previous command was move forward and there is an obstacle and the latest rotation was to the right then turn to the left.

This analysis suggests the definition of four internal states:

- S_{AL}: The robot is moving forward and the latest rotation was to the left.
- S_{AR}: The robot is moving forward and the latest rotation was to the right.
- S_{RR}: The robot is turning to the right.
- S_{RL}: The robot is turning to the left.

With those internal states the behavior of the robot control circuit is defined by the state transition graph of Fig. 4.10b (Moore model).

To define the combinational circuit of Fig. 4.10a the internal states of Fig. 4.10b must be encoded. For example:

$$S_{\mathrm{AR}}: \ q_0q_1 = 00, S_{\mathrm{RR}}: q_0q_1 = 01, S_{\mathrm{AL}}: \ q_0q_1 = 10, S_{\mathrm{RL}}: \ q_0q_1 = 11. \tag{4.4}$$

The following case instruction defines the combinational circuit function:

```
case q0q1 is
   when 00 => if OB = 0 then q0^Δ q1^Δ = 00;
              else q0^Δ q1^Δ = 11; end if;
              RR = 0; RL = 0;
   when 01 => if OB = 0 then q0^Δ q1^Δ = 00;
              else q0^Δ q1^Δ = 01; end if;
              RR = 1; RL = 0;
```

Table 4.1 Robot control circuit: next state table

Current state	Input: *OB*	Next state
S_{AR}	0	S_{AR}
S_{AR}	1	S_{RL}
S_{RR}	0	S_{AR}
S_{RR}	1	S_{RR}
S_{AL}	0	S_{AL}
S_{AL}	1	S_{RR}
S_{RL}	0	S_{AL}
S_{RL}	1	S_{RL}

Table 4.2 Robot control circuit: output table

Current state	Outputs: *RR RL*
S_{AR}	00
S_{RR}	10
S_{AL}	00
S_{RL}	01

```
   when 10 => if OB = 0 then q0^Δ q1^Δ = 10;
              else q0^Δ q1^Δ = 01; end if;
              RR = 0; RL = 0;
   when 11 => if OB = 0 then q0^Δ q1^Δ = 10;
              else q0^Δ q1^Δ = 11; end if;
              RR = 0; RL = 1;
end case;
```

4.3.3 Next State Table and Output Table

Instead of defining the behavior of a sequential circuit with a state transition graph, another option is to use tables. Once the set of internal states is known, the specification of the circuit of Fig. 4.3 amounts to the specification of the combinational circuit, for example by means of two tables:

- A table (next state table) that defines the next internal state in function of the current state and of the external input values
- A table (output table) that defines the external output values in function of the current internal state (Moore model) or in function of the current internal state and of the external input values (Mealy model)

As an example, the state transition diagram of Fig. 4.10b can be described by Tables 4.1 and 4.2.

4.4 Bistable Components

Bistable components such as latches and flip-flops are basic building blocks of any sequential circuit. They are used to implement the memory block of Fig. 4.3 and to synchronize the circuit operations with an external clock signal (Sect. 4.2).

4.4.1 1-Bit Memory

A simple 1-bit memory is shown in Fig. 4.11a. It consists of two interconnected inverters. This circuit has two stable states. In Fig. 4.11b the first inverter input is equal to 0 so that the second inverter input is equal to 1 and its output is equal to 0. Thus this is a stable state. Similarly another stable state is shown in Fig. 4.11c. This circuit has the capacity to store a 1-bit data. It remains to define the way a particular stable state can be defined.

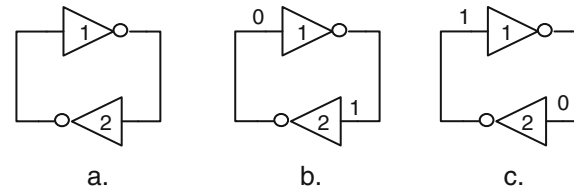
To control the state of the 1-bit memory, the circuit of Fig. 4.11a is completed with two tristate buffers controlled by an external *Load* signal (Fig. 4.12a):

- If $Load = 1$ then the circuit of Fig. 4.12a is equivalent to the circuit of Fig. 4.12b: the input D value (0 or 1) is transmitted to the first inverter input, so that the output P is equal to NOT(D) and $Q = \text{NOT}(P) = D$; on the other hand the output of the second inverter is disconnected from the first inverter input (buffer 2 in state Z, Sect. 2.4.3).
- If $Load = 0$ then the circuit of Fig. 4.12a is equivalent to the circuits of Fig. 4.12c and of Fig. 4.11a; thus it has two stable states; the value of Q is equal to the value of D just before the transition of signal *Load* from 1 to 0.

Observe that the two tristate buffers of Fig. 4.12a implement the same function as a 1-bit MUX2-1.

The circuit of Fig. 4.12a is a D-type latch. It has two inputs: a data input D and a control input *Load* (sometimes called *Enable*). It has two outputs: Q and $P = \overline{Q}$. Its symbol is shown in Fig. 4.13. Its working can be summarized as follows: when $Load = 1$ the value of D is sampled, and when $Load = 0$ this sampled value remains internally stored.

Fig. 4.11 1-Bit memory



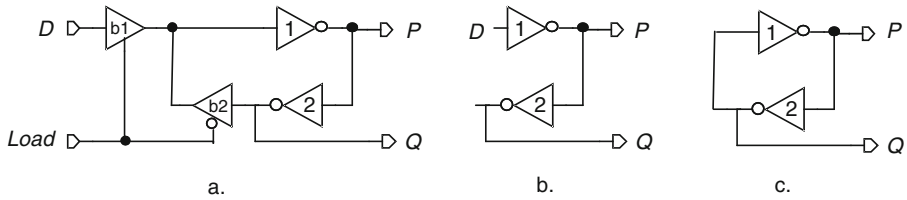
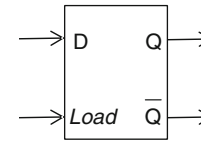


Fig. 4.12 D-type latch

Fig. 4.13 D-type latch symbol



Formally, a D-type latch could be defined as a sequential circuit with an external input D, an external output Q (plus an additional output $\overline{Q}$), and two internal states S_0 and S_1. The next state table and the output table are shown in Table 4.3.

However this circuit is not synchronized by an external clock signal; it is a so-called asynchronous sequential circuit. In fact, the external input *Load* could be considered as a clock signal input: the value of D is read and stored on each 1–0 (falling edge) of the *Load* signal so that the working of a D-type latch could be described by the equation $Q^{\Delta} = D$. Nevertheless, when $Load = 1$ then $Q = D$ (transparent state) and any change of D immediately causes the same change on Q, without any type of external synchronization.

Another way to control the internal state of the 1-bit memory of Fig. 4.11a is to replace the inverters by 2-input NAND or NOT gates. As an example, the circuit of Fig. 4.14a is an SR latch. It works as follows:

- If $S = R = 0$ then both NOR gates are equivalent to inverters (Fig. 4.14b) and the circuit of Fig. 4.14a is equivalent to a 1-bit memory (Fig. 4.11a).
- If $S = 1$ and $R = 0$ then the output of the first NOR is equal to 0, whatever the other input value, and the second NOR is equivalent to an inverter (Fig. 4.14c); thus $Q = 1$.
- If $S = 0$ and $R = 1$ then the output of the second NOR is equal to 0, whatever the other input value, and the first NOR is equivalent to an inverter (Fig. 4.14d); thus $Q = 0$.

To summarize, with $S = 1$ and $R = 0$ the latch is set to 1; with $S = 0$ and $R = 1$ the latch is reset to 0; with $S = R = 0$ the latch stores the latest written value. The combination $S = R = 1$ is not used (not allowed). The symbol of an SR latch is shown in Fig. 4.15.

Table 4.3 Next state table and output table of a D-type latch

Current state	*Load*	*D*	Next state	Output
S_0	0	–	S_0	0
S_0	1	0	S_0	0
S_0	1	1	S_1	0
S_1	0	–	S_1	1
S_1	1	0	S_0	1
S_1	1	1	S_1	1

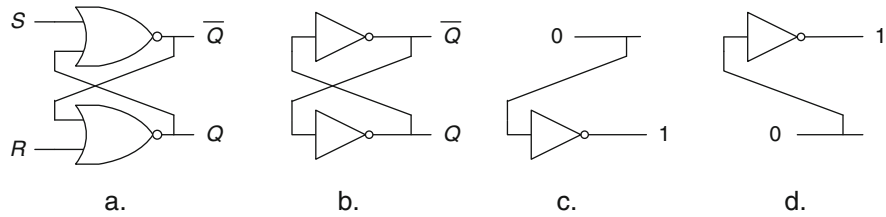
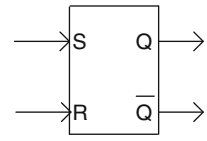


Fig. 4.14 SR latch

Fig. 4.15 Symbol of an SR latch



An SR latch is an asynchronous sequential circuit. Its state can only change on a rising edge of either S or R, and the new state is defined by the following equation: $Q^{\Delta} = S + \overline{R} \cdot Q$.

4.4.2 Latches and Flip-Flops

Consider again the sequential circuit of Fig. 4.3. The following question has not yet been answered: How the memory block is implemented? It has two functions: it stores the internal state and it synchronizes the operations by periodically updating the internal state under the control of a clock signal. In Sect. 4.4.1 a 1-bit memory component has been described, namely the D-type latch. A first option is shown in Fig. 4.16.

In Fig. 4.16 the memory block is made up of m D-type latches. The clock signal is used to periodically load new values within this m-bit memory. However, this circuit would generally not work correctly. The problem is that when $clock = 1$ all latches are in transparent mode so that after a rising edge of *clock* the new values of $q_0, q_1, \ldots, q_{m-1}$ could modify the values of ${q_0}^{\Delta}, {q_1}^{\Delta}, \ldots, {q_{m-1}}^{\Delta}$ before *clock* goes back to 0. To work correctly the clock pulses should be shorter than the minimum propagation time of the combinational circuit. For that reason another type of 1-bit memory element has been developed.

A D-type flip-flop is a 1-bit memory element whose state can only change on a positive edge of its clock input. A possible implementation and its symbol are shown in Fig. 4.17. It consists of two D-type latches controlled by *clock* and NOT(*clock*), respectively, so that they are never in transparent

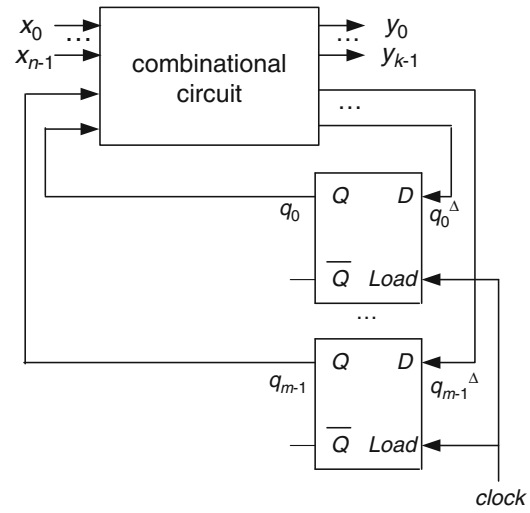


Fig. 4.16 Memory block implemented with latches

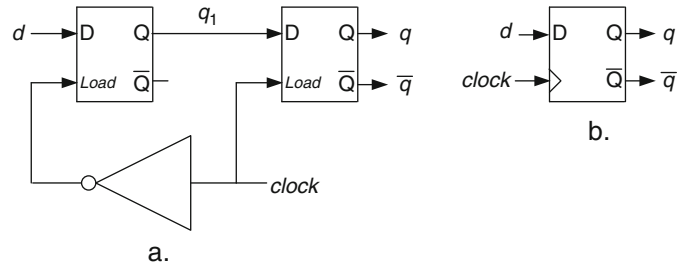


Fig. 4.17 D-type flip-flop

mode at the same time. When $clock = 0$ the first latch is in transparent mode so that $q_1 = d$ and the second latch stores the latest read value of q_1. When $clock = 1$ the first latch stores the latest read value of d and the second latch is in transparent mode so that $q = q_1$. Thus, the state q of the second latch is updated on the positive edge of *clock*.

Example 4.6 Compare the circuits of Fig. 4.18a, c: with the same input signals *Load* and *D* (Fig. 4.18b, d) the output signals *Q* are different. In the first case (Fig. 4.18b), the latch transmits the value of *D* to *Q* as long as *Load* = 1. In the second case (Fig. 4.18d) the flip-flop transmits the value of *D* to *Q* on the positive edges of *Load*.

Flip-flops need more transistors than latches. As an example the flip-flop of Fig. 4.17 contains two latches. But circuits using flip-flops are much more reliable: the circuit of Fig. 4.19 works correctly even if the clock pulses are much longer than the combinational circuit propagation time; the only

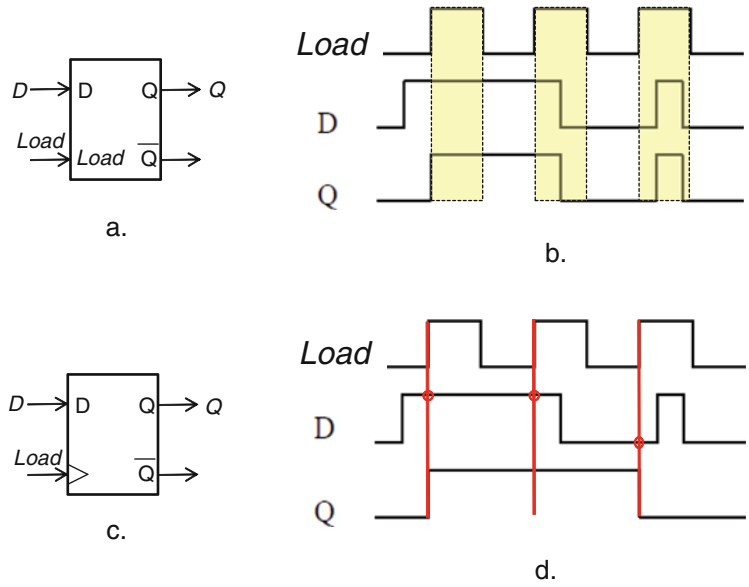


Fig. 4.18 Latch vs. flip-flop

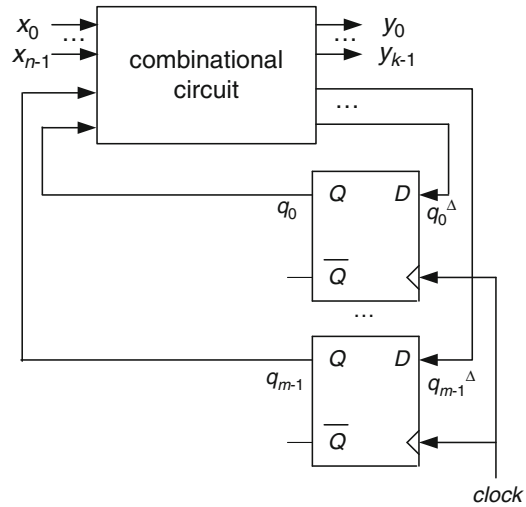


Fig. 4.19 Memory block implemented with flip-flops

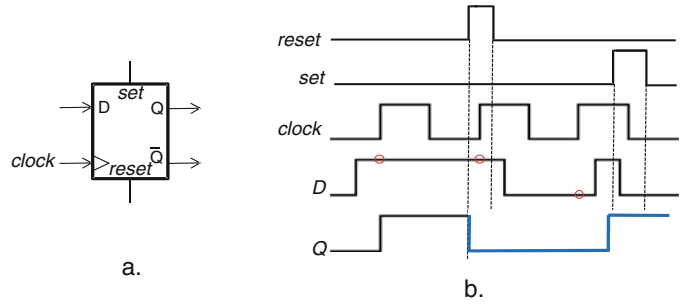


Fig. 4.20 D-type flip-flop with asynchronous inputs *set* and *reset*

timing condition is that the clock period must be greater than the combinational circuit propagation time. For that reason flip-flops are the memory components that are used to implement the memory block of sequential circuits.

Comment 4.2
D-type flip-flops can be defined as synchronized sequential circuits whose equation is

$$Q^{\Delta} = D. \tag{4.5a}$$

Other types of flip-flops have been developed: SR flip-flop, JK flip-flop, and T flip-flop. Their equations are

$$Q^{\Delta} = S + \overline{R} \cdot Q, \tag{4.5b}$$

$$Q^{\Delta} = J \cdot \overline{Q} + \overline{K} \cdot Q, \tag{4.5c}$$

$$Q^{\Delta} = T \cdot \overline{Q} + \overline{T} \cdot Q. \tag{4.5d}$$

Flip-flops are synchronous sequential circuits. Thus, (4.5a–4.5d) define the new internal state Q^{Δ} that will substitute the current value of Q on an active edge (positive or negative depending on the flip-flop type) of *clock*. Inputs D, S, R, J, K, and T are sometimes called synchronous inputs because their values are only taken into account on active edges of *clock*. Some components also have asynchronous inputs. The symbol of a D-type flip-flop with asynchronous inputs *set* and *reset* is shown in Fig. 4.20a. As long as $set = reset = 0$, it works as a synchronous circuit so that its state Q only changes on an active edge of *clock* according to (4.5a). However, if at some moment $set = 1$ then, independently of the values of *clock* and D, Q is immediately set to 1, and if at some moment $reset = 1$ then, independently of the values of *clock* and D, Q is immediately reset to 0. An example of chronogram is shown in Fig. 4.20b. Observe that the asynchronous inputs have an immediate effect on Q and have priority with respect to *clock* and D.

4.5 Synthesis Method

All the concepts necessary to synthesize a sequential circuit have been studied in the preceding sections. The starting point is a state transition graph or equivalent next state table and output table. Consider again the robot control system of Sect. 4.3.2. It has four internal states S_{AR}, S_{RR}, S_{RL}, and S_{AL} and it is described by the state transition graph of Fig. 4.10b or by a next state table (Table 4.1) and an output table (Table 4.2). The output signal values are defined according to the Moore model.

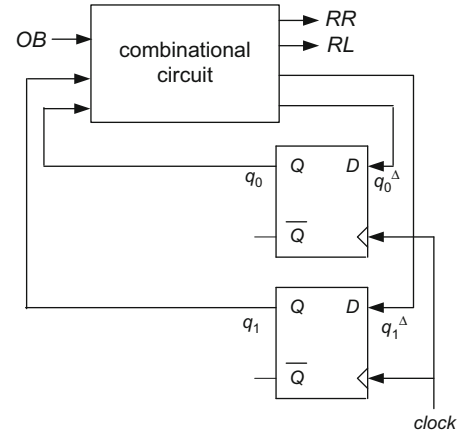


Fig. 4.21 Robot control circuit with D-type flip-flops

Table 4.4 Next state functions q_1^{Δ} and q_0^{Δ}

Current state $q_1\ q_0$	Input: *OB*	Next state $q_1^{\Delta}\ q_0^{\Delta}$
00	0	00
00	1	11
01	0	00
01	1	01
10	0	10
10	1	01
11	0	10
11	1	11

Table 4.5 External output functions *RR* and *RL*

Current state $q_1\ q_0$	Outputs: *RR RL*
00	00
01	10
10	00
11	01

The general circuit structure is shown in Fig. 4.3. In this example there is an external input *OB* ($n = 1$) and two external outputs *RR* and *RL* ($k = 2$), and the four internal states can be encoded with two variables q_0 and q_1 ($m = 2$). Thus, the circuit of Fig. 4.10a is obtained.

To complete the design, a first operation is to choose an encoding of the four states S_{AR}, S_{RR}, S_{RL}, and S_{AL} with two binary variables q_0 and q_1. Use for example the encoding of (4.4).

Another decision to be taken is the structure of the memory block. This point has already been analyzed in Sect. 4.4.2. The conclusion was that the more reliable option is to use flip-flops, for example D-type flip-flops (Fig. 4.17). The circuit that implements the robot control circuit is shown in Fig. 4.21 (Fig. 4.19 with $n = 1$, $k = 2$, and $m = 2$).

To complete the sequential circuit design it remains to define the functions implemented by the combinational circuit. From Tables 4.1 and 4.2, from the chosen internal state encoding (4.4), and from the D-type flip-flop specification (4.5a), Tables 4.4 and 4.5 that define the combinational circuit are deduced.

The implementation of a combinational circuit defined by truth tables has been studied in Chap. 2. The equations that correspond to Tables 4.4 and 4.5 are the following:

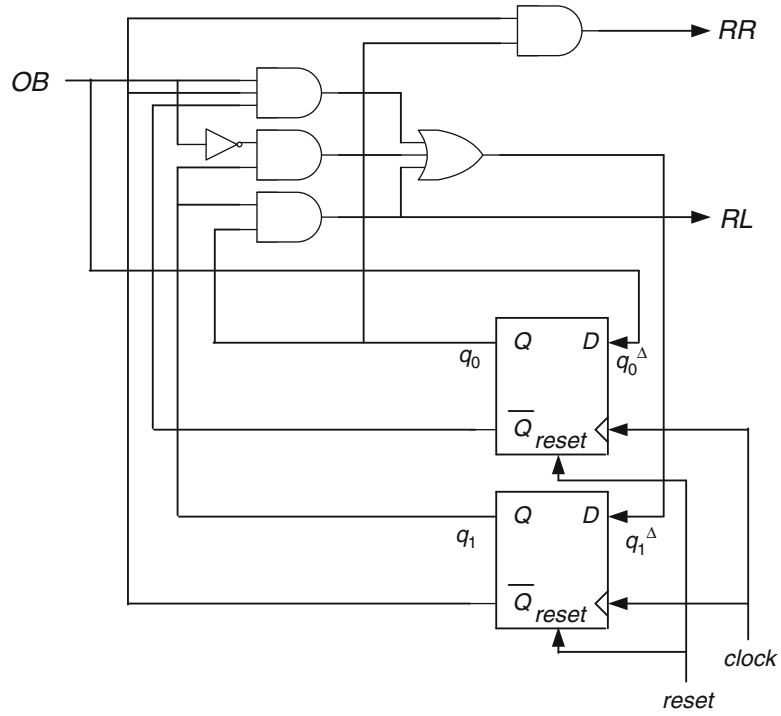


Fig. 4.22 Robot control circuit implemented with logic gates and D-type flip-flops

$$D_1 = q_1{}^{\Delta} = \overline{q}_1 \,.\, \overline{q}_0 \,.\, OB + q_1 \,.\, \overline{OB} + q_1 \,.\, q_0, D_0 = q_0{}^{\Delta} = OB,$$
$$RR = \overline{q}_1 . q_0, RL = q_1 . q_0.$$

The corresponding circuit is shown in Fig. 4.22.

Comments 4.3

- Flip-flops generally have two outputs Q and $\overline{Q}$ so that the internal state variables y_i are available under normal and under inverted form, and no additional inverters are necessary.

- An external asynchronous *reset* has been added. It defines the initial internal state ($q_1 \cdot q_0 = 00$, that is, state S_{AR}). In many applications it is necessary to set the circuit to a known initial state. Furthermore, to test the working of a sequential circuit it is essential to know its initial state.

  As a second example consider the state transition graph of Fig. 4.23, in this case a Mealy model. Its next state and output tables are shown in Table 4.6.
  The three internal states can be encoded with two internal state variables q_1 and q_0, for example

$$S_0 : q_1 q_0 = 00, S_1 : q_1 q_0 = 01, S_2 : q_1 q_0 = 10. \tag{4.6}$$

The circuit structure is shown in Fig. 4.24. According to Tables 4.5 and 4.6, the combinational circuit is defined by Table 4.7.

The equations that correspond to Table 4.7 are the following:

$$q_1^{\Delta} = q_0 \; + \; \overline{a}, \;\; q_0^{\Delta} = \; \overline{q_1} \cdot \overline{q_0} \cdot a, \;\; z = q_1 \cdot a.$$

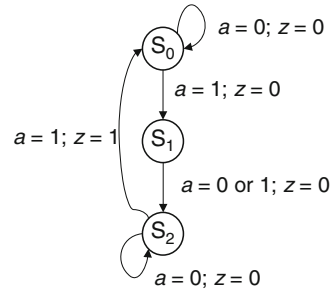


Fig. 4.23 Sequential circuit defined by a state transition graph (Mealy model)

Table 4.6 Next state and output tables (Mealy model)

Current state	a	Next state	z
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_2	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_0	1

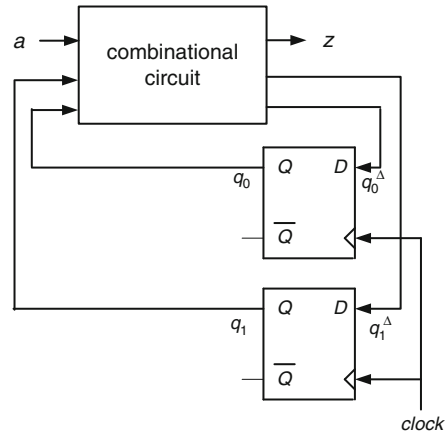


Fig. 4.24 Circuit structure

Table 4.7 Combinational circuit: truth table

Current state $q_1\ q_0$	Input: a	Next state $q_1^{\Delta}\ q_0^{\Delta}$	Output: z
00	0	00	0
00	1	01	0
01	0	10	0
01	1	10	0
10	0	10	0
10	1	00	1
11	0	–	–
11	1	–	–

4.6 Sequential Components

This section deals with particular sequential circuits that are building blocks of larger circuits, namely registers, counters, and memory blocks.

4.6.1 Registers

An n-bit register is a set of n D-type flip-flops or latches controlled by the same clock signal. They are used to store n-bit data. A register made up of n D-type flip-flops with asynchronous *reset* is shown in Fig. 4.25a and the corresponding symbol in Fig. 4.25b.

This parallel register is a sequential circuit with 2^n states encoded by an n-bit vector $q = q_{n-1}\, q_{n-2} \ldots q_0$ and defined by the following equations:

$$q^{\Delta} = IN, OUT = q. \tag{4.7}$$

Some registers have an additional OE (Output Enable) asynchronous control input (Fig. 4.26). This permits to connect the register output to a bus without additional tristate buffers (they are included in the register). In this example the control input is active when it is equal to 0 (active-low input, Fig. 2.29). For that reason it is called $\overline{OE}$ instead of OE.

Figures 4.25 and 4.26 are two examples of parallel registers. Other registers can be defined. For example: a set of n latches controlled by a *load* signal, instead of flip-flops, in which case $OUT = IN$ (transparent state) when $load = 1$. Thus they should not be used within feedback loops to avoid

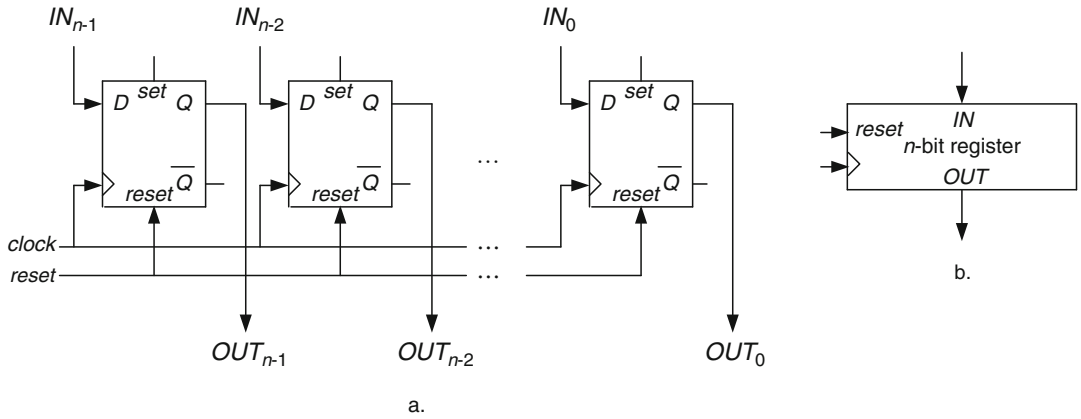


Fig. 4.25 n-Bit register

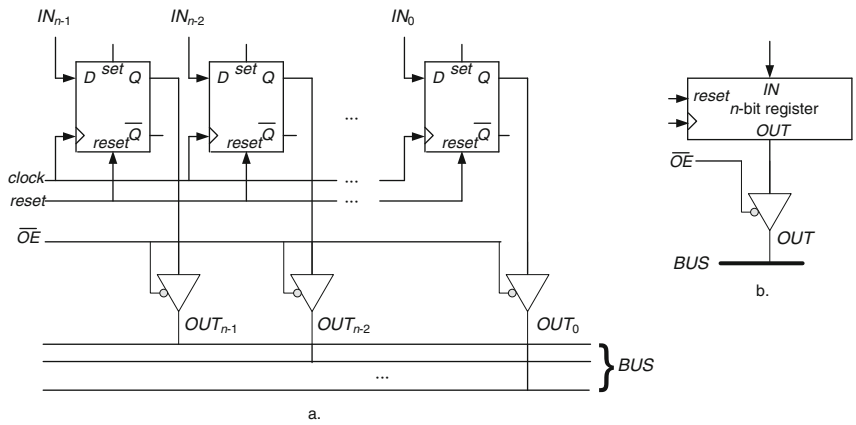


Fig. 4.26 n-Bit register with output enable (OE)

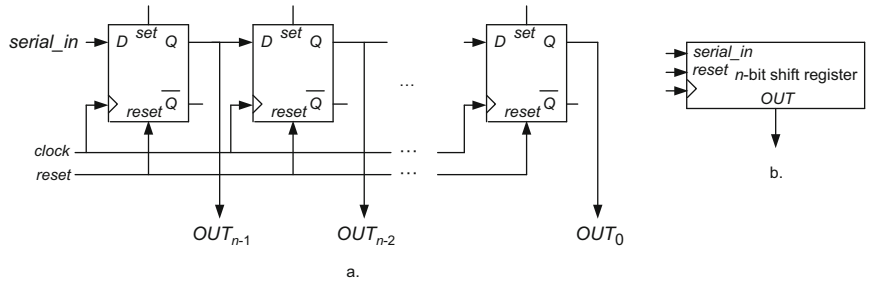


Fig. 4.27 Shift register

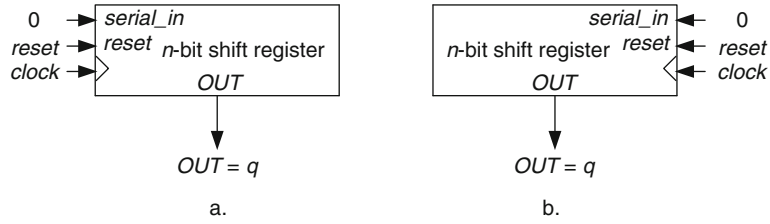


Fig. 4.28 Division and multiplication by 2

unstable states. Other examples of optional configurations: *clock* active-low or active-high, asynchronous *set* or *reset*, and *OE* active-low or active-high.

Shift registers are another type of commonly used sequential components. As parallel registers they consist of a set of n D-type flip-flops controlled by the same clock signal, so that they store an n-bit vector, but furthermore they can shift the stored data by one position to the right (to the left) at each clock pulse. An example of shift register is shown in Fig. 4.27. It has a serial input *serial_in* and a parallel output *OUT*. At each clock pulse a new bit is inputted, the stored word is shifted by one position to the right, and the last (least significant) bit is lost. This shift register is a sequential circuit with 2^n states encoded by an n-bit vector $q = q_{n\text{-}1}\, q_{n\text{-}2} \ldots q_0$ and defined by the following equations:

$$q_{n-1}{}^{\Delta} = serial\_in, q_i{}^{\Delta} = q_{i+1} \forall i = 0 \text{ to } n-2,\ OUT = q. \tag{4.8}$$

Shift registers have several applications. For example, assume that the current state q represents an n-bit natural. Then a shift to the right with $serial\_in = 0$ (Fig. 4.28a) amounts to the integer division of q by 2, and a shift to the left with $serial\_in = 0$ (Fig. 4.28b) amounts to the multiplication of q by 2 mod 2^n. For example, if $n = 8$ and the current state q is 10010111 (151 in decimal) then after a shift to the right $q = 01001011$ ($75 = \lfloor 151/2 \rfloor$ in decimal) and after a shift to the left $q = 00101110$ ($46 = 302 \bmod 256$ in decimal).

There are several types of shift registers that can be classified according to (among others)

- The shift direction: shift to the left, shift to the right, bidirectional shift, cyclic to the left, cyclic to the right
- The input type: serial or parallel input
- The output type: serial or parallel output

In Fig. 4.29 a 4-bit bidirectional shift register with serial input and parallel output is shown. When $L/R = 0$ the stored word is shifted to the left and when $L/R = 1$ the stored word is shifted to the right.

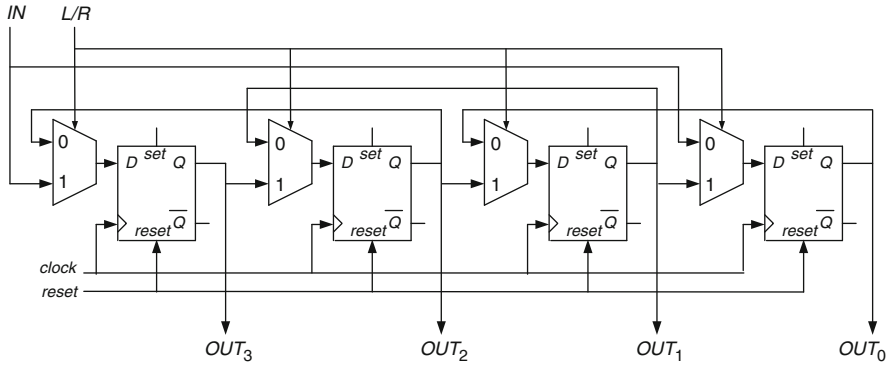


Fig. 4.29 4-Bit bidirectional shift register with serial input and parallel output

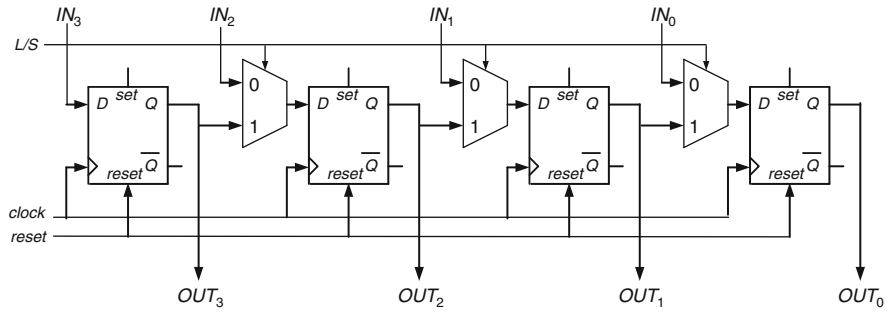


Fig. 4.30 4-Bit shift register with serial and parallel input and with parallel output

This shift register is a sequential circuit with 16 states encoded by a 4-bit vector $q = q_3\, q_2\, q_1\, q_0$ and defined by the following equations:

$$\begin{aligned} &q_3{}^{\Delta} = \overline{L/R} \cdot q_2 + L/R \cdot IN, q_i{}^{\Delta} = \overline{L/R} \cdot q_{i-1} + \ L/R \cdot q_{i+1} \forall i = 1 \text{ or } 2, \\ &q_0{}^{\Delta} = \overline{L/R} \cdot IN + L/R \cdot q_1, OUT = q. \end{aligned} \tag{4.9}$$

Another example is shown in Fig. 4.30: a 4-bit shift register, with serial input IN_3, parallel input $IN = (IN_3, IN_2, IN_1, IN_0)$, and parallel output OUT. When $L/S = 0$ the parallel input value is loaded within the register, and when $L/S = 1$ the stored word is shifted to the right and IN_3 is stored within the most significant register bit. This shift register is a sequential circuit with 16 states encoded by a 4-bit vector $q = q_3\, q_2\, q_1\, q_0$ and defined by the following equations:

$$q_3{}^{\Delta} = IN_3, q_i{}^{\Delta} = \overline{L/S} \cdot IN_i + L/S \cdot q_{i+1} \forall i = 0, 1, 2, \ \ OUT = q. \tag{4.10}$$

Shift registers with other control inputs can be defined. For example:

- *PL* (parallel load): When active, input bits IN_{n-1}, IN_{n-2}, …, IN_0 are immediately loaded in parallel, independently of the clock signal (asynchronous load).
- *CE* (clock enable): When active the *clock* signal is enabled; when nonactive the *clock* signal is disabled and, in particular, there is no shift.
- $\overline{OE}$ (output enable): When equal to 0 all output buffers are enabled and when equal to 1 all output buffers are in high impedance (state *Z*, disconnected).

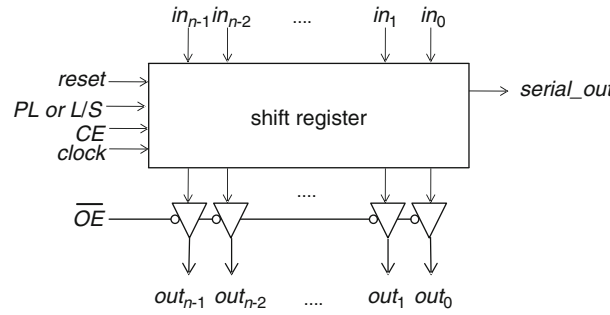


Fig. 4.31 Symbol of a shift register with control inputs PL or L/S, CE, and $\overline{OE}$

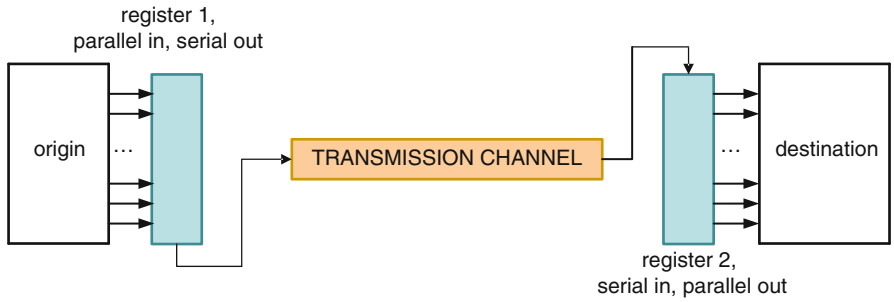


Fig. 4.32 Parallel-to-serial and serial-to-parallel conversion

Figure 4.31 is the symbol of a shift register with parallel input *in*, serial output *serial_out* and parallel output *out*, negative output enable control signal $\overline{OE}$ (to enable the parallel output), clock enable control signal (CE), and asynchronous *reset* input. With regard to the load and shift operations two options are considered: (1) with PL (asynchronous load): a new data is immediately stored when $PL = 1$ and the stored data is synchronously shifted to the right on a clock pulse when $CE = 1$; (2) with L/S (synchronous load): a new data is stored on a clock pulse when $L/S = 0$ and $CE = 1$, and the stored data is synchronously shifted to the right on a clock pulse when $L/S = 1$ and $CE = 1$.

Apart from arithmetic operations (multiplication and division by 2) shift registers are used in other types of applications. One of them is the parallel-to-serial and serial-to-parallel conversion in data transmission systems. Assume that a system called "origin" must send n-bit data to another system called "destination" using for that a 1-bit transmission channel (Fig. 4.32). The solution is a parallel-in serial-out shift register on the origin side and a serial-in parallel-out shift register on the destination side. To transmit a data, it is first loaded within register 1 (parallel input); then it is serially shifted out of the register 1 (serial output), it is transmitted on the 1-bit transmission channel, and it is shifted into register 2 (serial input); when all n bits have been transmitted the transmitted data is read from register 2 (parallel output).

Another application of shift registers is the recognition of sequences. Consider a sequential circuit with a 1-bit input *in* and a 1-bit output *out*. It receives a continuous string of bits and must generate an output $out = 1$ every time that the six latest received bits $in(t)$ $in(t-1)$ $in(t-2)$ $in(t-3)$ $in(t-4)$ $in(t-5)$ are 100101. A solution is shown in Fig. 4.33: a serial-in parallel-out shift register that stores the five values $in(t-1)$ $in(t-2)$ $in(t-3)$ $in(t-4)$ $in(t-5)$ and generates $out = 1$ when $in(t)$ $in(t-1)$ $in(t-2)$ $in(t-3)$ $in(t-4)$ $in(t-5) = 100101$.

Another example is shown in Fig. 4.34. This circuit has an octal input $in = in_2\ in_1\ in_0$ and a 1-bit output *out*. It receives a continuous string of digits and must generate an output $out = 1$ every time

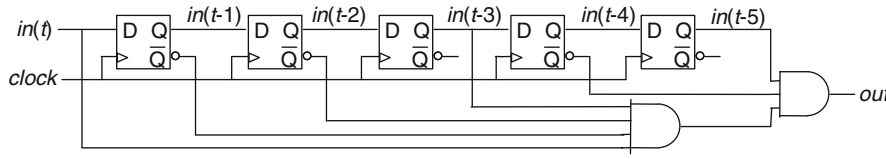


Fig. 4.33 Detection of sequence 100101

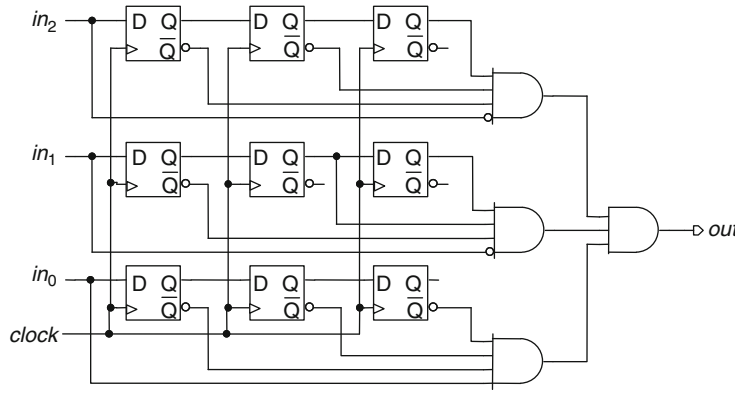


Fig. 4.34 Detection of sequence 1026

that the four latest received digits $in(t)$ $in(t-1)$ $in(t-2)$ $in(t-3)$ are 1026 (in binary 001 000 010 110). The circuit of Fig. 4.34 consists of three 3-bit shift registers that detect the 1-bit sequences $in_2 = 0001$, $in_1 = 0011$, and $in_0 = 1000$, respectively. An AND3 output gate generates $out = 1$ every time that the three sequences are detected.

4.6.2 Counters

Counters constitute another family of commonly used sequential components. An m-state counter (or mod m counter) is a Moore sequential circuit without external input and with an n-bit output q that is also its internal state and represents a natural belonging to the set $\{0, 1, \ldots, m-1\}$. At each clock pulse the internal state is increased or decreased. Thus the next state equation is

$$q^{\Delta} = (q+1) \bmod m (\text{up counter}) \text{ or } q^{\Delta} = (q-1) \bmod m (\text{down counter}). \tag{4.11}$$

Thus counters generate cyclic sequences of states. In the case of a mod m up counter the generated sequence is $\ldots 0\ 1 \ldots m-2\ m-1\ 0\ 1 \ldots$.

Definitions 4.1

- An n-bit binary up counter has $m = 2^n$ states encoded according to the binary numeration system. If $n = 3$ it generates the following sequence: 000 001 010 011 100 101 110 111 000 001 ....
- An n-bit binary down counter has $m = 2^n$ states encoded according to the binary numeration system. If $n = 3$ it generates the following sequence: 000 111 110 101 100 011 010 001 000 111 ....
- A binary coded decimal (BCD) up counter has ten states encoded according to the binary numeration system (BCD code). It generates the following sequence: 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 0000 0001 .... A BCD down counter is defined in a similar way.

Fig. 4.35 *n*-Bit up counter

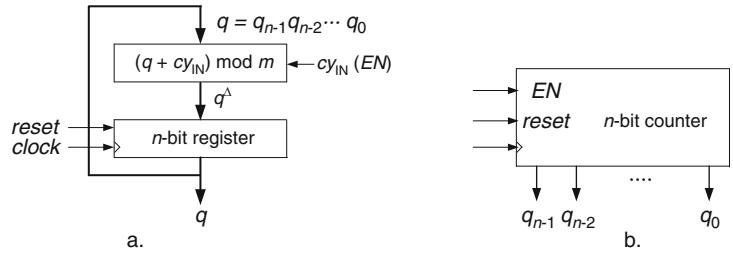
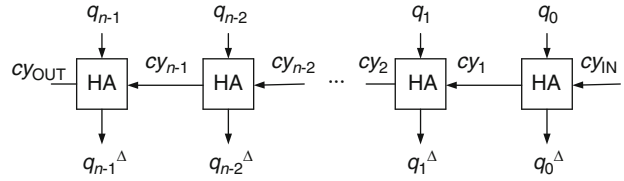


Fig. 4.36 *n*-Bit half adder



- An n-bit Gray counter has $m = 2^n$ states encoded in such a way that two successive states differ in only one position (one bit). For example, with $n = 3$, a Gray counter sequence is 000 010 110 100 101 111 011 001 000 010 ....
- Bidirectional counters have a control input *U/D* (up/down) that defines the counting direction (up or down).

The general structure of a counter is a direct consequence of its definition. If m is an n-bit number, then an m-state up counter consists of an n-bit register that stores the internal state q and of a modulo m adder that computes $(q + 1)$ mod m. In Fig. 4.35a a 1-operand adder (also called half adder) with a carry input cy_{IN} is used. The carry input can be used as an enable (*EN*) control input:

$$q^{\Delta} = EN \cdot [(q+1) \bmod m] + \overline{EN} \cdot q. \tag{4.12}$$

The corresponding symbol is shown in Fig. 4.35b.

If $m = 2^n$ then the mod m adder of Fig. 4.35a can be implemented by the circuit of Fig. 4.36 that consists of n 1-bit half adders. Each of them computes

$$q_i{}^{\Delta} = q_i \oplus cy_i, cy_{i+1} = q_i \cdot cy_i. \tag{4.13}$$

Observe that cy_{OUT} could be used to enable another counter so as to generate a $2n$-bit counter ($2^{2n} = m^2$ states) with two n-bit counters.

Example 4.7 According to (4.13) with $cy_0 = cy_{\text{IN}} = 1$ the equations of a 3-bit up counter are

$$q_0{}^{\Delta} = q_0 \oplus 1 = \overline{q_0}, q_1{}^{\Delta} = q_1 \oplus q_0, q_2{}^{\Delta} = q_2 \oplus q_1 \cdot q_0,$$

to which corresponds the circuit of Fig. 4.37.

Apart from *reset* and *EN* (Fig. 4.35) other control inputs can be defined, for example *OE* (output enable) as in the case of parallel registers (Fig. 4.26). An additional state output *TC* (terminal count) can also be defined: it is equal to 1 if, and only if, the current state $q = m - 1$. This signal is used to interconnect counters in series. If $m = 2^n$ then (Figs. 4.36 and 4.37) $TC = cy_{\text{OUT}}$.

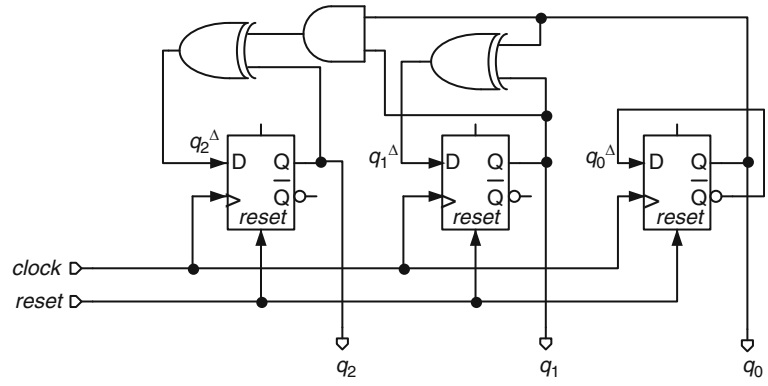


Fig. 4.37 3-Bit up counter

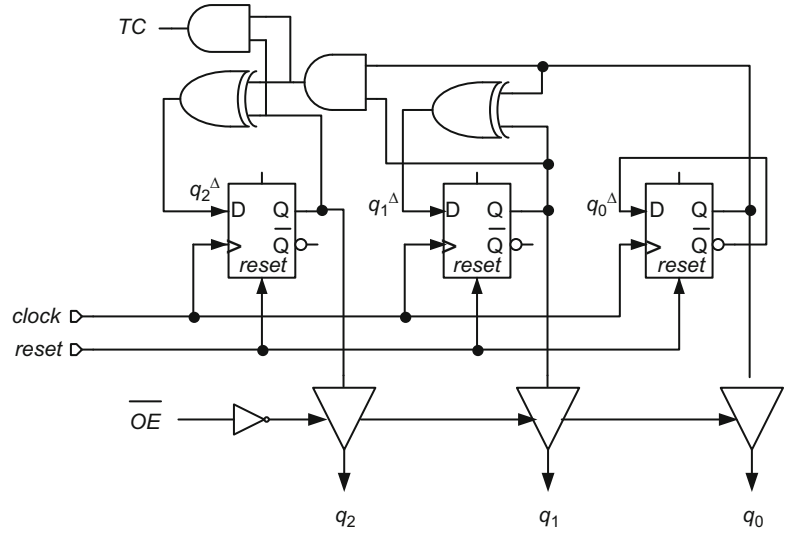


Fig. 4.38 3-Bit up counter with active-low *OE* and with *TC*

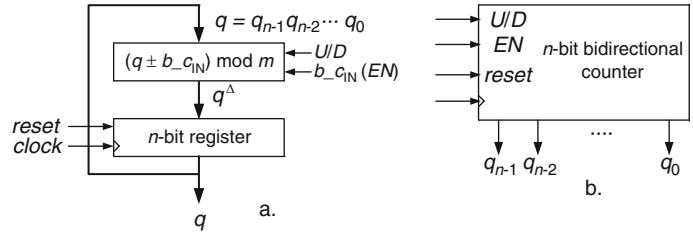


Fig. 4.39 Bidirectional *n*-bit counter

Example 4.8 In Fig. 4.38 an active-low *OE* control input and a *TC* output are added to the counter of Fig. 4.37. $TC = 1$ when $q = 7$ ($q_2 = q_1 = q_0 = 1$).

To implement a bidirectional (up/down) counter, the adder of Fig. 4.35 is replaced by an adder-subtractor (Fig. 4.39a). An *U/D* (up/down) control input permits to choose between addition ($U/D = 0$) and subtraction ($U/D = 1$). Input b_c_{IN} is an incoming carry or borrow that can be used to enable the counter. Thus

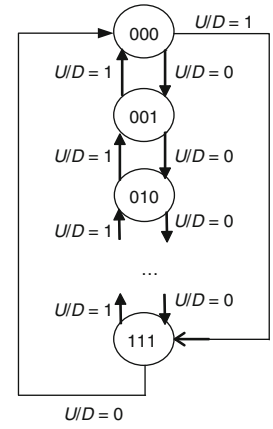
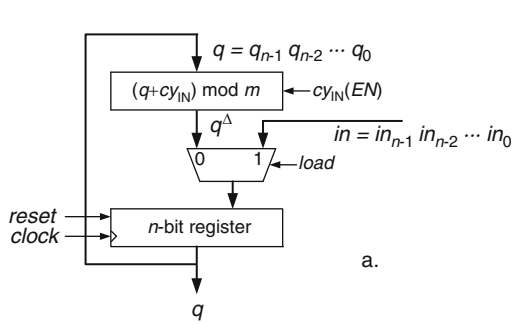


Fig. 4.40 State transition graph of a bidirectional 3-bit counter



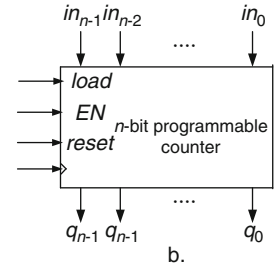


Fig. 4.41 Counter with parallel load

$$q^{\Delta} = EN \cdot \overline{U/D} \cdot [(q+1) \bmod m] + EN \cdot U/D \cdot [(q-1) \bmod m] + \overline{EN} \cdot q. \tag{4.14}$$

The corresponding symbol is shown in Fig. 4.39b.

As an example, the state transition graph of Fig. 4.40 defines a 3-bit bidirectional counter without *EN* control input ($b\_c_{\mathrm{IN}} = 1$).

In some applications it is necessary to define counters whose internal state can be loaded from an external input. Examples of applications are programmable timers and microprocessor program counters. An example of programmable counter with parallel load is shown in Fig. 4.41a. An *n*-bit MUX2-1 permits to write into the state register either q^{Δ} or an external input *in*. If $load = 0$ it works as an up counter, and when $load = 1$ the next internal state is *in*. Thus

$$q^{\Delta} = EN \cdot \overline{load} \cdot [(q+1) \bmod m] + EN \cdot load \cdot in + \overline{EN} \cdot q. \tag{4.15}$$

The corresponding symbol is shown in Fig. 4.41b.

Comment 4.3
Control inputs *reset* and *load* permit to change the normal counter sequence:

- If the counter is made up of flip-flops with *set* and *reset* inputs, an external *reset* command can change the internal state to any value (depending on the connection of the external *reset* signal to

individual flip-flop *set* or *reset* inputs), but always the same value, and this operation is asynchronous.
- The *load* command permits to change the internal state to any value defined by the external input *in*, and this operation is synchronous.

In Fig. 4.42 a counter with both asynchronous and synchronous reset is shown: the *reset* control input sets the internal state to 0 in an asynchronous way while the *synch_reset* input sets the internal state to 0 in a synchronous way.

Some typical applications using counters are now described. A first application is the implementation of timers. Consider an example: Synthesize a circuit with a 1-bit output z that generates a positive pulse on z every 5 s. Assume that a 1 kHz oscillator is available. The circuit is shown in Fig. 4.43. It consists of the oscillator and of a mod 5000 counter with state output *TC* (terminal count). The oscillator period is equal to 1 ms. Thus a mod 5000 counter generates a *TC* pulse every 5000 ms that is 5 s.

A second application is the implementation of systems that count events. As an example a circuit that counts the number of 1s in a binary sequence is shown in Fig. 4.44a. It is assumed that the binary sequence is synchronized by a *clock* signal. This circuit is an up counter controlled by the same *clock* signal. The binary sequence is inputted to the *EN* control input and the counter output gives the *number* of 1s within the input sequence. Every time that a 1 is inputted, the counter is enabled and one unit is added to *number*. An example is shown in Fig. 4.44b.

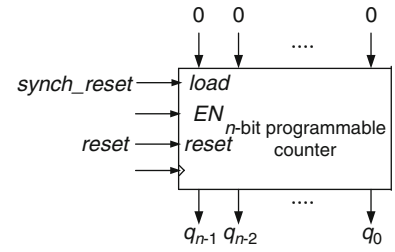


Fig. 4.42 Asynchronous and synchronous reset

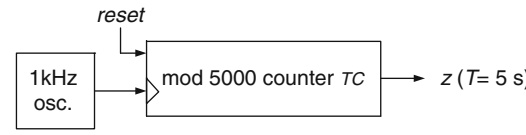


Fig. 4.43 Timer

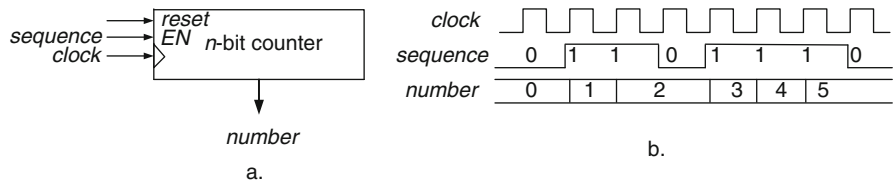


Fig. 4.44 Number of 1's counter

The 1-bit counter of Fig. 4.45a is a frequency divider. On each positive edge of *in*, connected to the clock input, the current value of $out = Q$ is replaced by its inverse $\overline{Q}$ (Fig. 4.45b). Thus, *out* is a square wave whose frequency is half the frequency of the input *in* frequency.

A last example of application is the implementation of circuits that generate predefined sequences. For example, to implement a circuit that repeatedly generates the sequence 10010101 a 3-bit mod 8 counter and a combinational circuit that computes a 3-variable switching function out_1 are used (Fig. 4.46a). Function out_1 (Table 4.8) associates a bit of the desired output sequence to each counter state. Another example is given in Fig. 4.46b and Table 4.8. This circuit repeatedly generates the sequence 100101. It consists of a mod 6 counter and a combinational circuit that computes a 3-variable switching function out_2 (Table 4.8).

The mod 6 counter of Fig. 4.46b can be synthesized as shown in Fig. 4.35a with $m = 6$ and $cy_{\text{IN}} = 1$. The combinational circuit that computes $(q + 1)$ mod 6 is defined by the following truth table (Table 4.9) and can be synthesized using the methods proposed in Chap. 2.

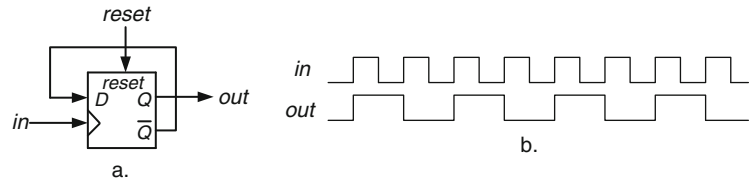


Fig. 4.45 Frequency divider by 2

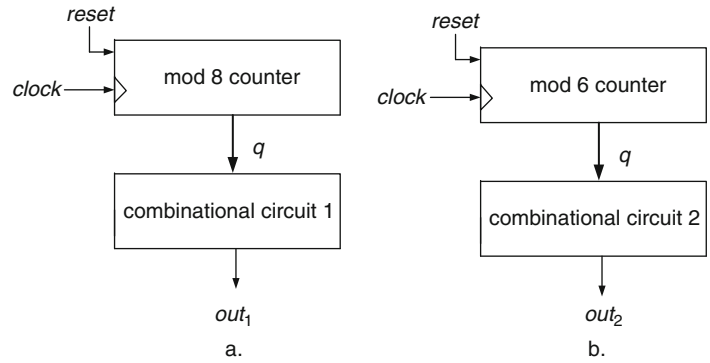


Fig. 4.46 Sequence generators

Table 4.8 Truth tables of out_1 and out_2

q	out_1	out_2
000	1	1
001	0	0
010	0	0
011	1	1
100	0	0
101	1	1
110	0	–
111	1	–

Table 4.9 Mod 6 addition

q	q^{Δ}
000	001
001	010
010	011
011	100
100	101
101	000
110	---
111	---

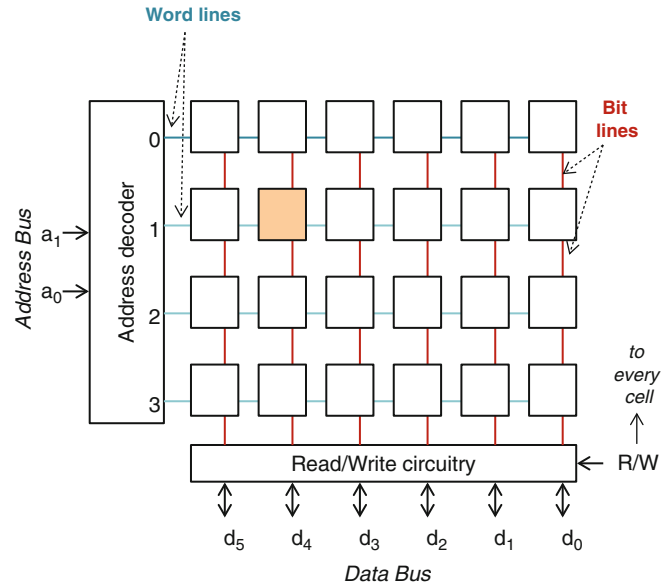


Fig. 4.47 Memory structure

4.6.3 Memories

Memories are essential components of any digital system. They have the capacity to store a large number of data. Functionally they are equivalent to a set of registers that can be accessed individually, either to write a new data or to read a previously stored data.

4.6.3.1 Types of Memories

A generic memory structure is shown in Fig. 4.47. It is an array of small cells; each of them stores a bit. This array is logically organized as a set of rows, where each row stores a word. In the example of Fig. 4.47 there are four words and each of them has six bits. The selection of a particular word, either to read or to write, is done by the address inputs. In this example, to select a word among four words, two bits a_1 and a_0 connected to an address bus are used. An address decoder generates the row selection signals (the word lines). For example, if $a_1\ a_0 = 10$ (2 in decimal) then word number 2 is selected. On the other hand, the bidirectional (input/output) data are connected to the bit lines. Thus, if word number 2 is selected by the address inputs, then d_5 is connected to bit number 5 of word number 2, d_4 is connected to bit number 4 of word number 2, and so on. A control input R/W (read/write) defines the operation, for example write if $R/W = 0$ and read if $R/W = 1$.

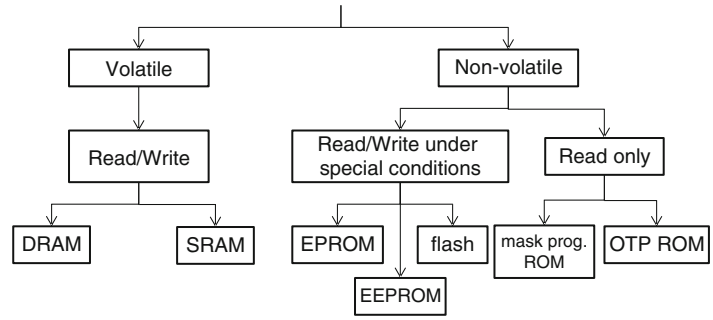


Fig. 4.48 Types of memories

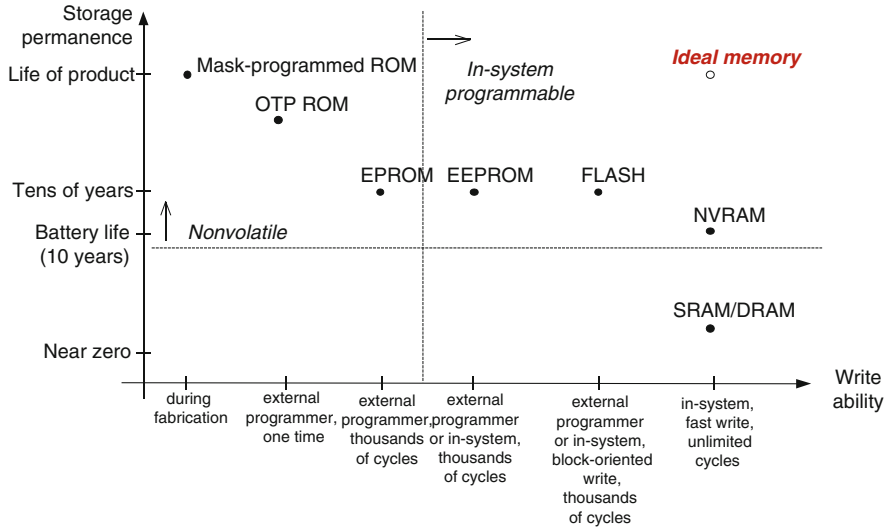


Fig. 4.49 Commercial memory types

A list of the main types of memories is given in Fig. 4.48. A first classification criterion is volatility: volatile memories lose their contents when the power supply is turned off while nonvolatile memories do not. Within nonvolatile memories there are read-only memories (ROM) and read/write memories. ROM are programmed either at manufacturing time (mask programmable ROM) or by the user but only one time (OTP = one-time programmable ROM). Other nonvolatile memories can be programmed several times by the user: erasable programmable ROM (EPROM), electrically erasable programmable ROM (EEPROM), and flash memories (a block-oriented EEPROM).

Volatile memories can be read and written. They are called random access memories (RAM) because the access time to a particular stored data does not depend on the particular location of the data (as a matter of fact a ROM has the same characteristic). There are two families: static RAM (SRAM) and dynamic RAM (DRAM).

The diagram of Fig. 4.49 shows different types of commercial memories classified according to their storage permanence (the maximum time without loss of information) and their ability to be programmed. Some memories can be programmed within the system to which they are connected (for example a printed circuit board); others must be programmed outside the system using a device called memory programmer. Observe also that EPROM, EEPROM, and flash memories can be reprogrammed a large number of times (thousands) but not an infinite number of times. With regard to the time necessary to write a data, SRAM and DRAM are much faster than nonvolatile memories. Nonvolatile RAM (NVRAM) is a battery-powered SRAM.

An ideal memory should have storage permanence equal to its lifetime, with the ability to be loaded as many times as necessary during its lifetime. The extreme cases are, on the one hand, mask programmable ROM that have the largest storage permanence but no reprogramming possibility and, on the other hand, static and dynamic RAM that have full programming ability.

4.6.3.2 Random Access Memories

RAMs are volatile memories. They lose their contents when the power supply is turned off. Their structure is the general one of Fig. 4.47. Under the control of the *R/W* control input, read and write operations can be executed: if the address bus $a = i$ and $R/W = 1$ then the data stored in word number i is transmitted to the data bus d; if the address bus $a = i$ and $R/W = 0$ then the current value of bus d is stored in word number i. Consider a RAM with n address bits that store m-bit words (in Fig. 4.47 $n = 2$ and $m = 6$). Its behavior is defined by the following piece of program:

```
i = conv(a);
if R/W = 0 then d = word(i); else word(i) = d; end if;
```

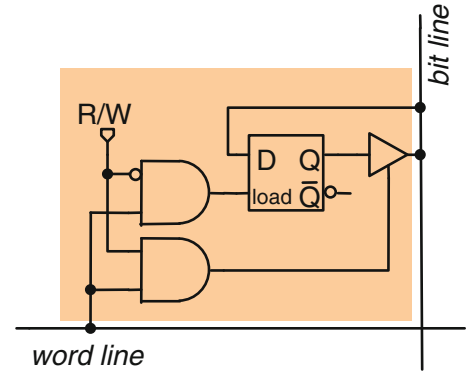
in which *conv* is a conversion function that translates an n-bit vector (an address) to a natural belonging to the interval 0 to $2^n - 1$ (this is the function of the address decoder of Fig. 4.47) and *word* is a vector of 2^n m-bit vectors (an array).

A typical SRAM cell is shown in Fig. 4.50. It consists of a D-type latch, some additional control gates, and a tristate buffer. The word line is an output of the address decoder and the bit line is connected to the data bus through the read/write circuitry (Fig. 4.47). When the word line is equal to 1 and $R/W = 0$ then the *load* input is equal to 1, the tristate output buffer is in high impedance (state *Z*, disconnected) and the value of the bit line connected to *D* is stored within the latch. When the word line is equal to 1 and $R/W = 1$ then the *load* input is equal to 0 and the value of *Q* is transmitted to the bit line through the tristate buffer.

Modern SRAM chips have a capacity of up to 64 megabits. Their read time is between 10 and 100 nanoseconds, depending on their size. Their power consumption is smaller than the power consumption of DRAM chips.

An example of very simple dynamic RAM (DRAM) cell is shown in Fig. 4.51a. It is made up of a very small capacitor and a transistor used as a switch. When the word line is selected, the cell capacitor is connected to the bit line through the transistor. In the case of a write operation, the bit line is connected to an external data input and the electrical charge stored in the cell capacitor is proportional to the input logic level (0 or 1). This electrical charge constitutes the stored information. However this information must be periodically refreshed because, in the contrary case, it would be

Fig. 4.50 SRAM cell



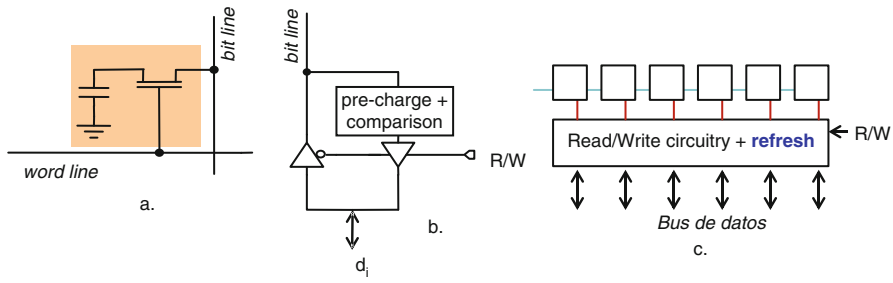


Fig. 4.51 DRAM cell

quickly lost due to the leakage currents. In the case of a read operation, the bit line is connected to a data output. The problem is that the cell capacitor is much smaller than the bit line equivalent capacitor, so that when connecting the cell capacitor to the bit line, the stored electrical charge practically disappears. Thus, the read operation is destructive. Some additional electronic circuitry is used to sense the very small voltage variations on the bit line when a read operation is executed: before the connection of the bit line to a cell capacitor, it is pre-charged with an intermediate value (between levels 0 and 1); then an analog comparator is used to sense the very small voltage variation on the bit line in order to decide whether the stored information was 0 or 1 (Fig. 4.51b). Once the stored information has been read (and thus destroyed) it is rewritten into the original memory location. The data bus interface of Fig. 4.51c includes the analog comparators (one per bit line) as well as the logic circuitry in charge of rewriting the read data. To refresh the memory contents, all memory locations are periodically read (and thus rewritten).

Modern DRAM chips have a capacity of up to 2 gigabits that is a much larger capacity than SRAM chips. On the other hand they are slower than SRAM and have higher power consumption.

4.6.3.3 Read-Only Memories

Within ROM a distinction must be done between mask programmable ROM whose contents are programmed at manufacturing time and programmable ROM (PROM) that can be programmed by the user, but only one time. Other names are one-time programmable (OTP) or write-once memories.

Their logic structure (Fig. 4.52) is also an array of cells. Each cell may connect, or not, a word line to a bit line. In the case of a mask programmable ROM the programming consists in drawing some of the word-line-to-bit-line connections in the mask that corresponds to one of the metal levels (Sect. 7.1). In the case of a user programmable ROM, all connections are initially enabled and some of them can be disabled by the user (fuse technologies) or none of them is previously enabled and some of them can be enabled by the user (anti-fuse technologies).

4.6.3.4 Reprogrammable ROM

Reprogrammable ROMs are user programmable ROM whose contents can be reprogrammed several times. Their logic structure is the same as that of non-reprogrammable ROMs but the word-line-to-bit-line connections are floating-gate transistors instead of metal connections.

There are three types of reprogrammable ROM:

- EPROM: Their contents are erased by exposing the chip to ultraviolet (UV) radiation; for that, the chip must be removed from the system (for example the printed circuit board) in which it is used; the chip package must have a window to let the UV light reach the floating-gate transistors; an external programmer is used to (re)program the memory.

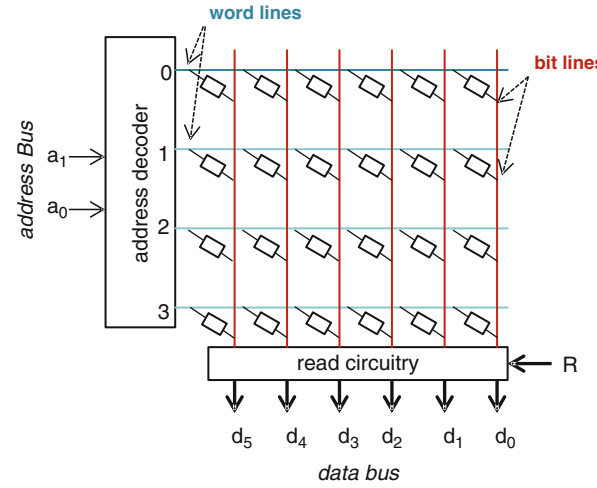


Fig. 4.52 Read-only memory structure

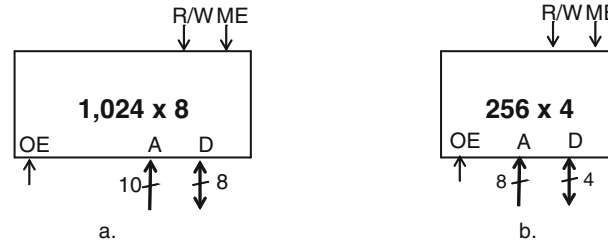


Fig. 4.53 Implementation of a 1 kB memory

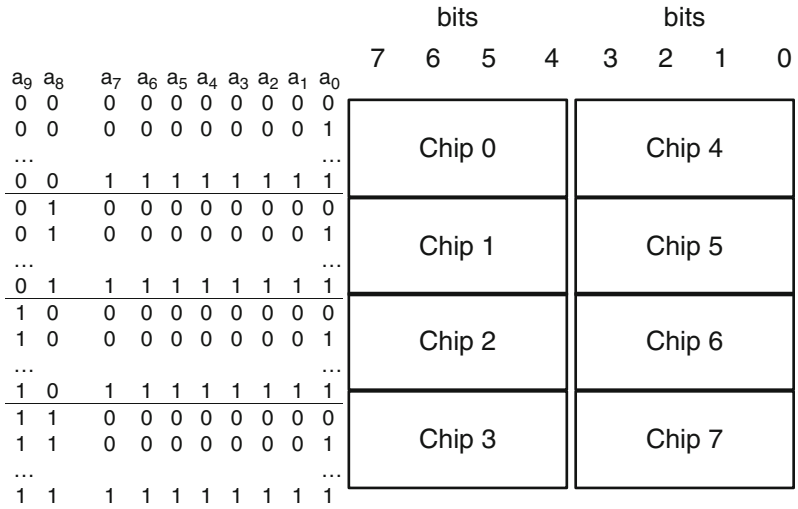
- EEPROM: Their contents are selectively erased, one word at a time, using a specific higher voltage; the chip must not be removed from the system; the (re)programming circuitry is included within the chip.
- Flash memories: This is an EEPROM-type memory with better performance; in particular, block operations instead of one-word-at-a-time operations are performed; they are used in many applications, for example pen drives, memory cards, solid-state drives, and many others.

4.6.3.5 Example of Memory Bank

Memory banks implement large memories with memory chips whose capacity is smaller than the desired capacity. As an example, consider the implementation of the memory of Fig. 4.53a with a capacity of 1 kB (1024 words, 8 bits per word) using for that the memory chip of Fig. 4.53b that can store 256 4-bit words. Thus, eight memory chips must be used ($1024 \cdot 8 = (256 \cdot 4) \cdot 8$).

Let $a_9\ a_8\ a_7 \ldots a_0$ be the address bits of the memory (Fig. 4.53a) to be implemented. The 1024-word addressing space is decomposed into four blocks of 256 words, using for that bits a_9 and a_8. Each block of 256 words is implemented with two chips working in parallel (Fig. 4.54). To select one of the four blocks the *OE* (output enable) control inputs are used:

Fig. 4.54 Address space



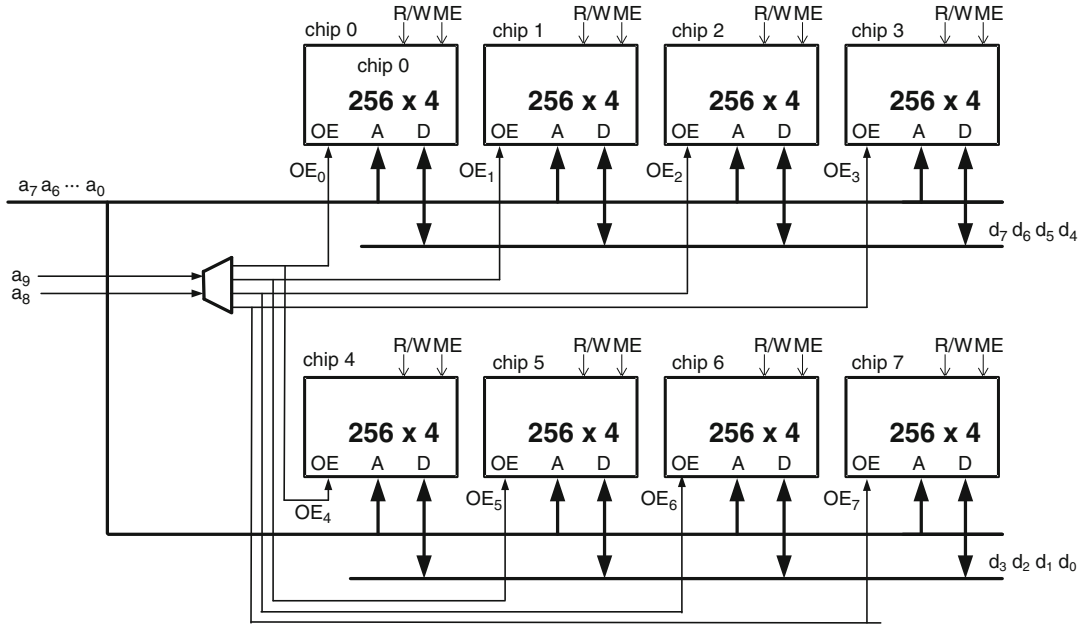


Fig. 4.55 Memory bank

$$\begin{aligned} OE_0 = OE_4 = \overline{a_9} \cdot \overline{a_8},\ OE_1 = OE_5 = \overline{a_9} \cdot a_8, \\ OE_2 = OE_6 = a_9 \cdot \overline{a_8}\ , OE_0 = OE_4 = a_9 \cdot a_9. \end{aligned} \tag{4.16}$$

The complete memory bank is shown in Fig. 4.55. A 2-to-4 address decoder generates the eight output enable functions (4.16).

More information about memories can be found in classical books such as Weste and Harris (2010) or Rabaey et al. (2003).

4.7 Sequential Implementation of Algorithms

In Sect. 2.8 the relation between algorithms (programming language structures) and combinational circuits has been commented. This relation exists between algorithms and digital circuits in general, not only combinational circuits. As a matter of fact the understanding and the use of this relation is a basic aspect of this course.

Some systems are sequential by nature because their specification includes an explicit or implicit reference to successive time intervals. Some examples have been seen before: generation and detection of sequences (Examples 4.1 and 4.2), control of sequences of events (Sects. 4.1 and 4.3.2), data transmission (Fig. 4.32), timers (Fig. 4.43), and others. However algorithms without any time reference can also be implemented by sequential circuits.

4.7.1 A First Example

As a first example of synthesis of a sequential circuit from an algorithm, a circuit that computes the integer square root of a natural is designed. Given a natural x it computes $r = \lfloor \sqrt{x} \rfloor$ where $\lfloor a \rfloor$ stands for the greatest natural smaller than or equal to a. The following algorithm computes a set of successive pairs (r, s) such that $s = (r + 1)^2$.

Algorithm 4.1

```
r0 = 0; s0 = 1;
for i in 0 to N loop
   si+1 = si + 2·(ri + 1) + 1;
   ri+1 = ri + 1;
end loop;
```

It uses the following relation:

$$(r+2)^2 = ((r+1)+1)^2 = (r+1)^2 + 2\cdot(r+1) + 1 = s + 2\cdot(r+1) + 1.$$

Assume that $x > 0$. Initially $s_0 = 1 \leq x$. Then execute the loop as long as $s_i \leq x$. When $s_i \leq x$ and $s_{i+1} > x$ then

$$r_{i+1}{}^2 = (r_i + 1)^2 = s_i \leq x \text{ and } (r_{i+1} + 1)^2 = s_{i+1} > x.$$

Thus r_{i+1} is the greatest natural smaller than or equal to $\sqrt{x}$ so that $r = r_{i+1}$. The following naïve algorithm computes r.

Algorithm 4.2 Square Root

```
r = 0; s = 1;
while s ≤ x loop
    s = s + 2·(r+1) + 1;
    r = r + 1;
end loop;
root = r;
```

Table 4.10 Example of square root computation

r	s	$s \leq 47$
0	1	*true*
1	4	*true*
2	9	*true*
3	16	*true*
4	25	*true*
5	36	*true*
6	49	*false*

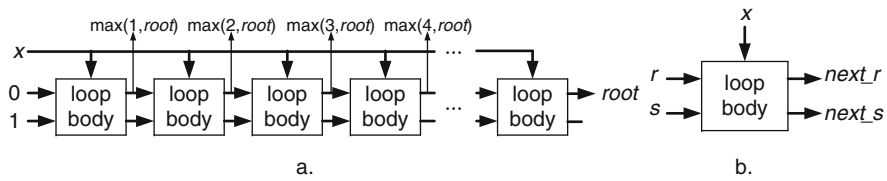


Fig. 4.56 Square root implementation: iterative circuit

As an example, if $x = 47$ the successive values of r and s are given in Table 4.10. The result is $r = 6$ (the first value of r such that condition $s \leq x$ does not hold true).

This is obviously not a good algorithm. The number of steps is equal to the square root r. Thus, for great values of $x \cong 2^n$, the number of steps is $r \cong 2^{n/2}$. It is used for didactic purposes.

Algorithm 4.2 is a loop so that a first option could be an iterative combinational circuit (Sect. 2.8.3). In this case the number of executions of the loop body is not known in advance; it depends on a condition ($s \leq x$) computed before each loop body execution. For that reason the algorithm must be slightly modified: once the condition stops holding true, the values of s and r do not change any more.

Algorithm 4.3 Square Root, Version 2

```
r = 0; s = 1;
loop
   if s ≤ x then
      next_s = s + 2·(r+1) + 1; next_r = r + 1;
   else
      next_s = s; next_r = r;
   end if;
   s = next_s; r = next_r;
end loop;
root = r;
```

The corresponding iterative circuit is shown in Fig. 4.56a. The iterative cell (Fig. 4.56b) executes the loop body, and the connections between adjacent cells implement the instructions $s = next\_s$ and $r = next\_r$.

The first output of circuit number i is either number i or the square root of x, and once the square root has been computed the first output value does not change any more and is equal to the square root r. Thus the number of cells must be equal to the maximum value of r. But this is a very large number. As an example, if $n = 32$, then $x < 2^{32}$ and $r < 2^{16} = 65{,}536$ so that the number of cells must be equal to 65,535, obviously a too large number of cells.

A better idea is a sequential implementation. It takes advantage of the fact that a sequential circuit not only implements combinational functions but also includes memory elements and, thanks to the use of a synchronization signal, permits to divide the time into time intervals to which corresponds the execution of different operations. In this example two memory elements store r and s, respectively, and the time is divided into time intervals to which are associated groups of operations:

- First time interval (initial value of the memory elements):

```
r = 0; s = 1;
```

- Second, third, fourth, ... time interval:

```
if s ≤ x then
   next_s = s + 2·(r+1) + 1; next_r = r + 1;
 else
   next_s = s; next_r = r;
end if;
```

In the following algorithm a variable *end* has been added; it detects the end of the square root computation.

Algorithm 4.4 Square Root, Version 3

```
r = 0; s = 1; -- (initial values of memory elements)
loop
    if s ≤ x then next_s = s + 2·(r+1) + 1;
       next_r = r + 1; end = FALSE;
    else
       next_s = s; next_r = r; end = TRUE;
    end if;
    s = next_s; r = next_r; --synchronization
end loop;
root <= r;
```

As long as $s \leq x$, *end* is equal to *false*. When for the first time $s > x$ then *end* is equal to *true* and the values of r, s, and *end* will not change any more. The synchronization input *clock* will be used to modify the value of r and s by replacing their current values by their next values *next_r* and *next_s*.

The sequential circuit of Fig. 4.57 implements Algorithm 4.4. Two registers store r and s. On $reset = 1$ the initial values are loaded: $r = 0$, and $s = 1$. The combinational circuit implements functions *next_r*, *next_s* and *end* as defined by the loop body:

```
if s ≤ x then next_s = s + 2·(r+1) + 1;
   next_r = r + 1; end = FALSE;
else
   next_s = s; next_r = r; end = TRUE;
end if;
```

On each *clock* pulse the current values of r and s are replaced by *next_r* and *next_s*, respectively.

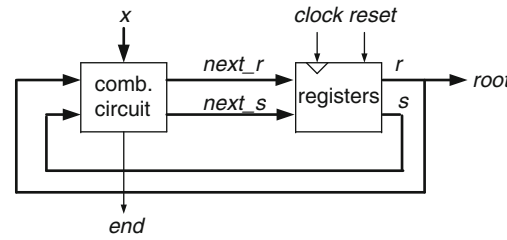


Fig. 4.57 Square root implementation: sequential circuit

Comment 4.4
Figure 4.57 is a sequential circuit whose internal states are all pairs $\{(r, s), r < 2^{n/2}, s = (r + 1)^2\}$ and whose input values are all natural $x < 2^n$. The corresponding transition state graph would have $2^{n/2}$ vertices and 2^n edges per vertex: for example ($n = 32$) 65,536 vertices and 4,294,967,296 edges per vertex. Obviously the specification of this sequential circuit by means of a graph or by means of tables doesn't make sense. This is an example of system that is described by an algorithm, not by an explicit behavior description.

4.7.2 Combinational vs. Sequential Implementation

In many cases an algorithm can be implemented either by a combinational or by a sequential circuit. In the case of the example of the preceding section (Sect. 4.7.1) the choice of a sequential circuit was quite evident. In general, what are the criteria that must be used?

Conceptually any well-defined algorithm, without time references, can be implemented by a combinational circuit. For that

- Execute the algorithm using all input variable value combinations and store the corresponding output variable values in a table.
- Generate the corresponding combinational circuit.

Except in the case of very simple digital systems, this is only a theoretical proof that a combinational circuit could be defined. The obtained truth table generally is enormous so that this method would only make sense if the designer has an unbounded space to implement the circuit (for example unbounded silicon area, Chap. 7) and has an unbounded time to develop the circuit.

A better method is to directly translate the algorithm instructions to circuits as was done in Sect. 2.8, but even with this method the result might be a too large circuit as shown in the preceding section (Sect. 4.7.1).

In conclusion, there are algorithms that cannot reasonably be implemented by a combinational circuit. As already mentioned above, a sequential circuit has the capacity to implement switching functions but also the capacity to store data within memory elements. Furthermore, the existence of a synchronization signal permits to divide the time into time intervals and to assign different time intervals to different operations.

In particular, in the case of loop instructions, the availability of memory elements allows to substitute N identical components by one component that iteratively executes the loop body, so that space (silicon area) is replaced by time. But this method is not restricted to the case of iterations. As an example, consider the following algorithm that consists of four instructions:

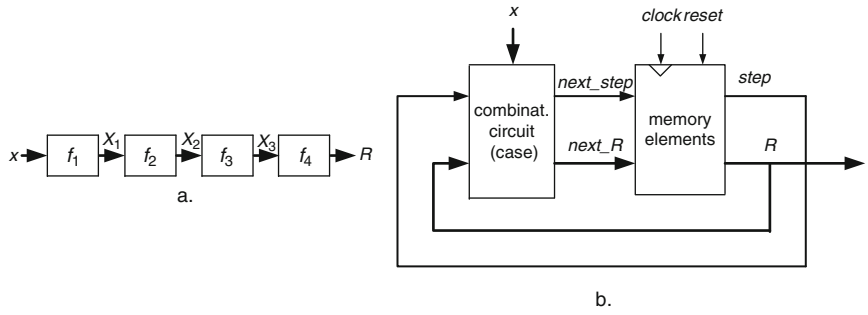


Fig. 4.58 Combinational vs. sequential implementations

```
0: X1 = f1(x);
1: X2 = f2(X1);
2: X3 = f3(X2);
3: R = f4(X3);
```

It can be implemented by a combinational circuit (Fig. 4.58a) made up of four components that implement functions f_1, f_2, f_3, and f_4, respectively. Assume that all data x, X_1, X_2, X_3, and R have the same number n of bits. Then the preceding algorithm could also be implemented by a sequential circuit. For that the algorithm is modified and a new variable *step*, whose values belong to {0, 1, 2, 3, 4}, is added:

```
step = 0; -- (initial value of the step identifier)
loop
    case step is
        when 0 => R = f1(x); step = 1;
        when 1 => R = f2(R); step = 2;
        when 2 => R = f3(R); step = 3;
        when 3 => R = f4(R); step = 4;
        when 4 => step = 4;
    end case;
end loop;
```

The sequential circuit of Fig. 4.58b implements the modified algorithm. It has two memory elements that store R and *step* (encoded) and a combinational circuit defined by the following *case* instruction:

```
case step is
  when 0 => next_R = f1(x); next_step = 1;
  when 1 => next_R = f2(R); next_step = 2;
  when 2 => next_R = f3(R); next_step = 3;
  when 3 => next_R = f4(R); next_step = 4;
  when 4 => next_step = 4;
end case;
```

When *reset* = 1 the initial value of *step* is set to 0. On each clock pulse R and *step* are replaced by *next_step* and *next_R*.

What implementation is better? Let c_i and t_i be the cost and computation time of the component that computes f_i. Then the cost C_{comb} and computation time T_{comb} of the combinational circuit are

$$C_{comb} = c_1 + c_2 + c_3 + c_4 \text{ and } T_{comb} = t_1 + t_2 + t_3 + t_4. \tag{4.17}$$

The cost C_{sequ} and the computation time T_{sequ} of the sequential circuit are equal to

$$C_{sequ} = C_{case} + C_{reg} \text{ and } T_{sequ} = 4 \cdot T_{clock}, \tag{4.18}$$

where C_{case} is the cost of the circuit that implements the combinational circuit of Fig. 4.58b, C_{reg} is the cost of the registers that store R and *step*, and T_{clock} is the *clock* signal period. The combinational circuit of Fig. 4.58b is a programmable resource that, under the control of the *step* variable, computes f_1, f_2, f_3, or f_4. There are two extreme cases:

- If no circuit part can be shared between some of those functions, then this programmable resource implicitly includes the four components of Fig. 4.58a plus some additional control circuit; its cost is greater than the sum $c_1 + c_2 + c_3 + c_4 = C_{comb}$ and T_{clock} must be greater than the computation time of the slowest component; thus $T_{sequ} = 4{\cdot}T_{clock}$ is greater than $T_{comb} = t_1 + t_2 + t_3 + t_4$.
- The other extreme case is when $f_1 = f_2 = f_3 = f_4 = f$; then the algorithm is an iteration; let c and t be the cost and computation time of the component that implements f; then $C_{comb} = 4{\cdot}c$, $T_{comb} = 4{\cdot}t$, $C_{sequ} = c + C_{reg}$, and $T_{sequ} = 4{\cdot}T_{clock}$ where T_{clock} must be greater than t; if the register cost is much smaller than c and if the clock period is almost equal to t then $C_{sequ} \cong c = C_{comb}/4$ and $T_{sequ} \cong 4{\cdot}t = T_{comb}$.

In the second case (iteration) the sequential implementation is generally better than the combinational one. In other cases, it depends on the possibility to share, or not, some circuit parts between functions f_1, f_2, f_3, and f_4.

In conclusion, algorithms can be implemented by circuits that consist of

- Memory elements that store variables and time interval identifiers (*step* in the preceding example)
- Combinational components that execute operations depending on the particular time interval, with operands that are internally stored variables and input signals

The circuit structure is shown in Fig. 4.59. It is a sequential circuit whose internal states correspond to combinations of variable values and step identifier values.

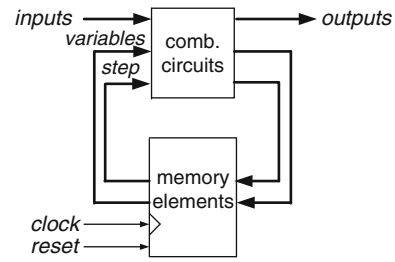


Fig. 4.59 Structure of a sequential circuit that implements an algorithm

4.8 Finite-State Machines

Finite-state machines (FSM) are algebraic models of sequential circuits. As a matter of fact the physical implementation of a finite-state machine is a sequential circuit so that part of this section repeats subjects already studied in previous sections.

4.8.1 Definition

Like a sequential circuit, a finite-state machine has input signals, output signals, and internal states. Three finite sets are defined:

- Σ (input states, input alphabet) is the set of values of the input signals.
- Ω (output states, output alphabet) is the set of values of the output signals.
- S is the set of internal states.

The working of the finite-state machine is specified by two functions f (next-state function) and h (output function):

- f: $S \times \Sigma \rightarrow S$ associates an internal state to every pair (internal state, input state).
- h: $S \times \Sigma \rightarrow \Omega$ associates an output state to every pair (internal state, input state).

Any sequential circuit can be modelled by a finite-state machine.

Example 4.9 A 3-bit up counter, with *EN* (count enable) control input, can be modelled by a finite-state machine: it has one binary input *EN*; three binary outputs q_2, q_1, and q_0; and eight internal states, so that

$$\begin{aligned}
\Sigma &= \{0,\ 1\},\\
\Omega &= \{000,\ 001,\ 010,\ 011,\ 100,\ 101,\ 110,\ 111\},\\
S &= \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7\},\\
f(s,\ 0) &= s \text{ and } f(s,\ 1) = s + 1 \bmod 8, \forall s \in S,\\
h(s,\ 0) &= h(s,\ 1) = \mathit{binary_encoded}(s), \forall s \in S,
\end{aligned}$$

where *binary_encoded*(s) is the binary representation of s. The corresponding sequential circuit is shown in Fig. 4.60: it consists of a 3-bit register and a combinational circuit that implements f, for example a multiplexer and a circuit that computes q + 1.

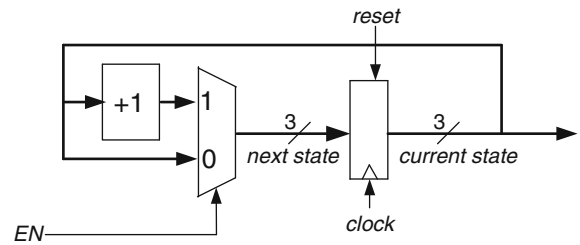


Fig. 4.60 3-Bit counter

Thus, finite-state machines can be seen as a formal way to specify the behavior of a sequential circuit. Nevertheless, they are mainly used to define the working of circuits that control sequences of operations rather than to describe the operations themselves.

The difference between the Moore and Mealy models has already been seen before (Sect. 4.3.1). In terms of finite-state machines, the difference is the output function h definition. In the case of the Moore model

$$h : S \rightarrow \Omega \tag{4.19}$$

and in the case of the Mealy model

$$h : S \times \Sigma \rightarrow \Omega. \tag{4.20}$$

In the first case the corresponding circuit structure is shown in Fig. 4.61. A first combinational circuit computes the next state in function of the current state and of the input. Assume that its propagation time is equal to t_1 seconds. Another combinational circuit computes the output state in function of the current state. Assume that its propagation time is equal to t_2 seconds. Assume also that the input signal comes from another synchronized circuit and is stable $t_{SUinput}$ seconds (SU means Set Up) after the active *clock* edge.

A chronogram of the signal values during a state transition is shown in Fig. 4.62. The register delay is assumed to be negligible so that the new current state value (register output) is stable at the beginning of the clock cycle. The output will be stable after t_2 seconds. The input is stable after $t_{SUinput}$ seconds ($t_{SUinput}$ could be the value t_2 of another finite-state machine). The next state will be stable $t_{SUinput} + t_1$ seconds later. In conclusion, the clock period must be greater than t_2 and $t_{SUinput} + t_1$:

$$T_{clock} > max\{t_{SUinput} + t_1, t_2\}. \tag{4.21}$$

This is an example of computation of the minimum permitted clock period and thus of the maximum clock frequency.

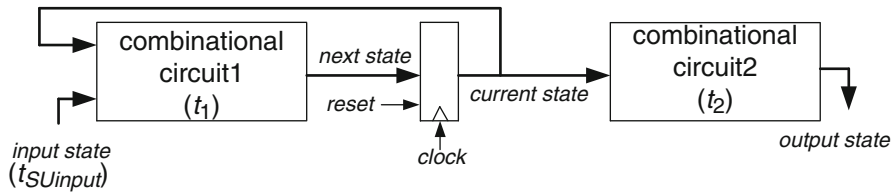


Fig. 4.61 Moore model: sequential circuit structure

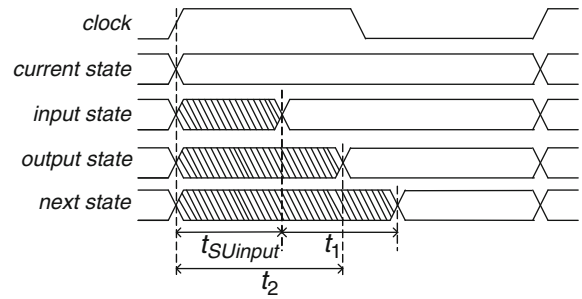


Fig. 4.62 Moore model: chronogram

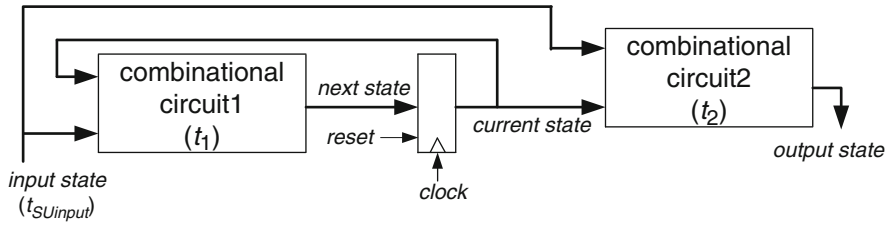


Fig. 4.63 Mealy model: sequential circuit structure

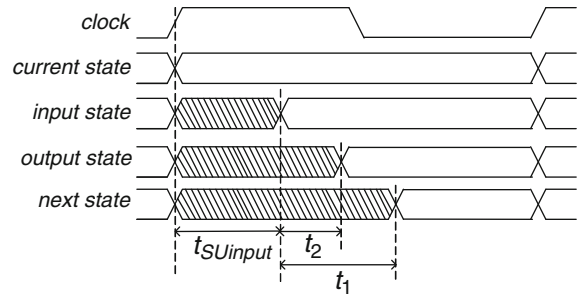


Fig. 4.64 Mealy model: chronogram

The circuit structure that corresponds to the Mealy model is shown in Fig. 4.63. A first combinational circuit computes the next state in function of the current state and of the input. Assume that its propagation time is equal to t_1 seconds. Another combinational circuit computes the output state in function of the current state and of the input. Assume that its propagation time is equal to t_2 seconds. Assume also that the input signal comes from another synchronized circuit and is stable $t_{SUinput}$ seconds (*SU* means Set Up) after the active *clock* edge.

A chronogram of the signal values during a state transition is shown in Fig. 4.64. As before, the register delay is assumed to be negligible so that the new current state value is stable at the beginning of the clock cycle. The input is stable after $t_{SUinput}$ seconds. The next state will be stable $t_{SUinput} + t_1$ seconds later and the output will be stable $t_{SUinput} + t_2$ seconds later. In conclusion, the clock period must be greater than $t_{SUinput} + t_2$ and $t_{SUinput} + t_1$:

$$T_{clock} > max\{t_{SUinput} + t_1, t_{SUinput} + t_2\}. \tag{4.22}$$

4.8.2 VHDL Model

All along this course a formal language (pseudo-code), very similar to VHDL, has been used to describe algorithms. In this section, complete executable VHDL definitions of finite-state machines are presented. An introduction to VHDL is given in Appendix A.

The structure of a Moore finite-state machine is shown in Fig. 4.61. It consists of three blocks: a combinational circuit that computes the next state, a combinational circuit that computes the output, and a register. Thus, a straightforward VHDL description consists of three processes, one for each block:

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_fsm.all;
entity MooreFsm is
port (
    clk, reset: in std_logic;
    x: in std_logic_vector(N-1 downto 0);
    y: out std_logic_vector(M-1 downto 0)
);
end MooreFsm;
architecture behavior of MooreFsm is
  signal current_state, next_state: state;
begin

  next_state_function: process(current_state, x)
  begin
      next_state <= F(current_state, x);
  end process next_state_function;
  synchronization: process(clk, reset)
  begin
    if reset = '1' then current_state <= S0;
    elsif clk'event and clk = '1' then
      current_state <= next_state;
    end if;
  end process synchronization;
  output_state: process(current_state)
  begin
     y <= H(current_state);
  end process output_state;

end behavior;
```

To understand the previous generic VHDL model it must be assumed that a package *my_fsm* has been previously defined. It includes

- The definition of the set of internal states, for example

```
type state is (S0, S1, S2, ...);
```

- The number N of input signals
- The number M of output signals
- The definition of the next state function F
- The definition of the output function H

Then, the *MooreFsm* entity is defined. Packages *ieee.std_logic_1164* (definition of the logic values and functions) and *my_fsm* are made visible and the input and output signals (ports) are defined, namely an N-bit input x and an M-bit output y. The architecture includes the declaration of two signals

current_state and *next_state* whose type *state* must have been defined in the package *my_fsm*. Then three processes are defined: *next_state_function* describes the first combinational circuit, *synchronization* corresponds to the register, and *output_state* describes the second combinational circuit. Observe the way the positive edge of *clk* is defined:

```
clk'event and clk = '1'
```

With regard to the sensitivity lists, the *next_state_function* and *output_state* processes describe combinational circuits and must be sensitive to all their input signals (*current_state* and *x* in the first circuit and *current_state* in the second). The process *synchronization* describes a register and is sensitive to *clk* and to the asynchronous input *reset* but it is not sensitive to the data input *next_state*: the register state can only change when $reset = 1$ or on an active edge of *clk*. It is interesting to observe that the *reset* signal has priority with respect to the *clk* synchronization signal:

```
if reset = '1' then current_state <= S0;
elsif clk'event and clk = '1' then ...
end if;
```

If $reset = 1$ then the synchronization signal has no effect.

Instead of defining F and H within a separate package, they could be directly defined within the processes, for example using *case* constructs (as in Example 4.5 or in Sect. 4.3.2). The processes *next_state_function* and *output_state* have the following structure:

```
next_state_function: process(current_state, x)
begin
 case current_state is
 when S0 =>
  <instructions that execute current_state <= F(S0,x)>;
 when S1 =>
  <instructions that execute current_state <= F(S1,x)>;
 when S2 =>
  <instructions that execute current_state <= F(S2,x)>;
 ...
 end case;
end process next_state_function;
output_state: process(current_state)
begin
 case current_state is
  when S0 => <instructions that execute y <= H(S0)>;
  when S1 => <instructions that execute y <= H(S1)>;
  when S2 => <instructions that execute y <= H(S2)>;
  ...
 end case;
end process output_state;
```

Example 4.10 The following VHDL program defines a 3-bit counter:

```
package my_fsm is
   type state is range 0 to 7;
end;
library ieee;
use ieee.std_logic_1164.all;
use work.my_fsm.all;
entity counter is
port (
   clk, reset, count_enable: in std_logic;
   y: out std_logic_vector(2 downto 0)
);
end counter;
architecture behavior of counter is
   signal current_state, next_state: state;
begin
   next_state_function: process(current_state, count_enable)
   begin
     case current_state is
       when 0 =>
         if count_enable = '1' then next_state <= 1; end if;
       when 1 =>
         if count_enable = '1' then next_state <= 2; end if;
       when 2 =>
         if count_enable = '1' then next_state <= 3; end if;
       when 3 =>
         if count_enable = '1' then next_state <= 4; end if;
       when 4 =>
         if count_enable = '1' then next_state <= 4; end if;
       when 5 =>
         if count_enable = '1' then next_state <= 6; end if;
       when 6 =>
         if count_enable = '1' then next_state <= 7; end if;
       when 7 =>
         if count_enable = '1' then next_state <= 0; end if;
   end case;
end process next_state_function;

synchronization: process(reset, clk)
begin
  if reset = '1' then current_state <= 0;
  elsif clk'event and clk = '1' then
    current_state <= next_state;
  end if;
 end process synchronization;
 output_state: process(current_state)
 begin
    case current_state is
```

```
      when 0 => y <= "000";
      when 1 => y <= "001";
      when 2 => y <= "010";
      when 3 => y <= "011";
      when 4 => y <= "100";
      when 5 => y <= "101";
      when 6 => y <= "110";
      when 7 => y <= "111";
 end case;
end process output_state;
end behavior;
```

The structure of a Mealy finite-state machine is shown in Fig. 4.63. The only difference with the circuit of Fig. 4.61 is that the output is a function of the current state and of the input: $y = H$ (*current_state*, x). The *output_state* process has the following structure:

```
output_state: process(current_state, x)
begin
 case current_state is
  when S0 => <instructions that execute y <= H(S0,x)>;
  when S1 => <instructions that execute y <= H(S1,x)>;
  when S2 => <instructions that execute y <= H(S2,x)>;
  ...
 end case;
end process output_state;
```

Comment 4.5
In the preceding generic structures, processes *next_state_function* and *synchronization* could be merged into one process *next_state_registered* that models the first combinational circuit and the register connected to its output (Figs. 4.61 and 4.63). The *next_state* signal is no longer necessary.

```
next_state_registered: process(reset, clk)
begin
  if reset = '1' then current_state <= S0;
  elsif clk'event and clk = '1' then
   case current_state is
     when S0 => <instructions that execute current_state <= F(S0, x)>;
     when S1 => <instructions that execute current_state <= F(S1, x)>;
     when S2 => <instructions that execute current_state <= F(S2, x)>;
         ...
   end case;
  end if;
end process next_state_registered;
```

This process is sensitive to *clk* and to the asynchronous input *reset* but is not sensitive to the data inputs *current_state* and x as the register state can only change when $reset = 1$ or on an active edge of *clk*. So, with regard to the circuit simulation, there is a difference between the three-process and two-process descriptions: in the first case, the process *next_state_function* is executed every time there is an event on signals *current_state* and x, and the *synchronization* process is executed every

time there is an event on signals *reset* and *clk*; in the second case, the process *next_state_registered* is executed every time there is an event on signals *reset* and *clk*.

4.9 Examples of Finite-State Machines

Two examples of circuits whose implementation includes a finite-state machine are described.

4.9.1 Programmable Timer

The first example is a programmable timer (Fig. 4.65a). It has three inputs:

- *start* is a binary signal.
- *reference* is a square wave signal generated by an external oscillator, with a known period $T_{reference}$.
- *time* is an *n*-bit number that expresses the timer delay as a multiple of $T_{reference}$.

The timer has a binary output *done*. It works as follows (Fig. 4.65b): on a positive edge of *start*, the value *k* of the input signal *time* is read and stored within the circuit; at the same time the output signal *done* goes down and remains at low level for about *k* times the period $T_{reference}$ of the signal *reference*.

The timer working is formally defined by the following algorithm:

Algorithm 4.5: Programmable Timer

```
loop
    done = 1;
    wait until start = 1;
        done = 0; count = time;
        loop
            wait until reference = 1;
            count = count - 1;
            if count = 0 then exit; end if;
        end loop;
end loop;
```

Algorithm 4.5 is an infinite loop. Initially *done* is set to 1. Then the timer waits for a positive edge of input *start*. When *start* = 1, the output *done* goes down and the value of the input *time* is stored within an internal variable *count*. Then the timer executes an internal loop: it waits for a positive edge

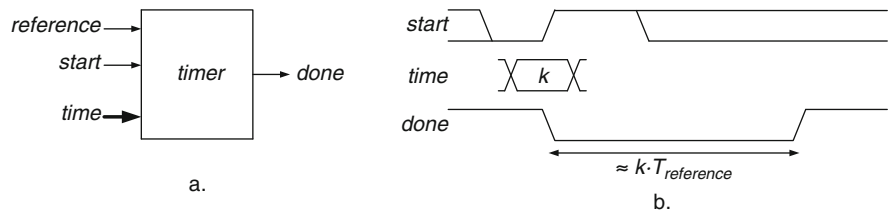
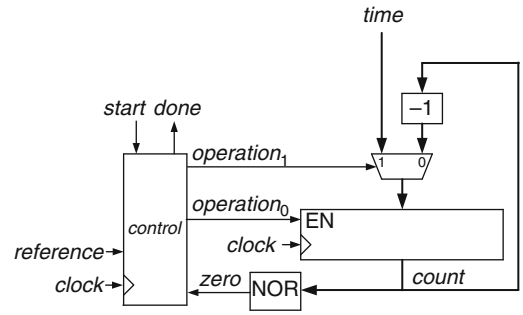
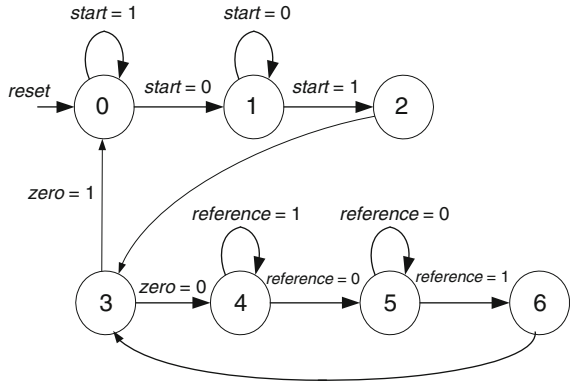


Fig. 4.65 Programmable timer





a.

state	operation	done
0	00	1
1	00	1
2	11	0
3	00	0
4	00	0
5	00	0
6	01	0

b.

Fig. 4.66 Implementation of Algorithm 4.5

Fig. 4.67 Finite-state machine that controls the sequence of operations

of the signal *reference*, then it decrements the value of *count*, it checks whether *count* is equal to 0 or not, and, if *count* = 0, it goes out of this internal loop (*exit* instruction). Then, it completes the main loop body execution (*end loop*), it goes back to the beginning of the main loop, and it sets *done* to 1.

An implementation of Algorithm 4.5 is shown in Fig. 4.66. An n-bit register stores the value of variable *count*. This register can be loaded with the value of the input signal *time* or with the output signal of an arithmetic circuit that computes *count* − 1. An n-input NOR gate detects when *count* = 0. A control unit defines the sequence of operations. It has three inputs: the external inputs *start* and *reference* and an internal signal *zero* generated by the NOR gate. It has two outputs: the external output *done* and a 2-bit internal control signal *operation*; the MUX2-1 control input is $operation_1$ and the *EN* register input is $operation_0$.

The control circuit can be modelled by the finite-state machine of Fig. 4.67. It is a Moore machine, with six internal states. Initially the finite-state machine waits for a positive edge of *start*: if *start* = 1 it waits for *start* being equal to 0 (state 0) and when *start* = 0 it waits for a positive edge of *start* (state 1). When in state 2, the value of *time* is stored in the register *count*. State 3 is a branch: if *count* is equal to 0 the machine goes back to state 0, and if *count* is not equal to 0 a positive edge of the signal *reference* must be detected: the control unit first waits for *reference* = 0 (state 4) and then it waits for *reference* = 1 (state 5). In state 6, *count* is decremented and the machine goes back to state 3.

The output function (Fig. 4.67b) is defined by a table. In states 0 and 1 *done* remains equal to 1, $EN = operation_0 = 0$, and the MUX2-1 control input $operation_1 = 0$ or 1 (don't care). In state 2 *time* must be stored in the register; thus $EN = operation_0 = 1$ and $operation_1 = 1$; *done* = 0. In states 3, 4, and 5 the register contents are not modified so that $operation_0 = 0$ and $operation_1 = 0$ or

1; *done* = 0. Finally, in state 6 the value of *count* is updated: $operation_0 = 1$ and $operation_1 = 0$; *done* = 0.

The following *next_state_registered* and *output_state* processes (two-process model, Comment 4.5) describe the control unit:

```
next_state_registered: process(reset, clk)
begin
 if reset = '1' then current_state <= 0;
 elsif clk'event and clk = '1' then
  case current_state is
    when 0 => if start = '0' then current_state <= 1; end if;
    when 1 => if start = '1' then current_state <= 2; end if;
    when 2 => current_state <= 3;
    when 3 => if zero = '1' then current_state <= 0;
              else current_state <= 4; end if;
    when 4 => if reference = '0' then current_state <= 5; end if;
    when 5 => if reference = '1' then current_state <= 6; end if;
    when 6 => current_state <= 3;
  end case;
 end if;
end process next_state_registered;
output_state: process(current_state)
begin
  case current_state is
    when 0 to 1 => operation <= "00"; done <= '1';
    when 2 =>      operation <= "11"; done <= '0';
    when 3 to 5 => operation <= "00"; done <= '0';
    when 6 =>      operation <= "01"; done <= '0';
  end case;
end process output_state;
```

This is a typical example of use of a finite-state machine. The timer is divided into two synchronous circuits: the rightmost part of Fig. 4.66 includes a register that stores an internal signal *count*, an arithmetic circuit (−1), and a logic circuit that computes a *zero* condition; the leftmost part is a control unit that receives an external command (*start*), another external input (*reference*), and a status signal (*zero*), and controls the working of the other part. The rightmost part is a (very simple) data path, that is to say a circuit made up of memory elements and computation resources that has the capacity to process data. It is controlled by a control unit that defines the sequence of operations executed by the data path. The division into data path and control unit is a very common design technique, and the modelling of control units is one of the main applications of finite-state machines.

Comment 4.6

A direct description of the complete programmable timer circuit, without an explicit partition into data path and control unit, could also be considered:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity timer is
```

```
port (
    start, clk, reset, reference: in std_logic;
    time: in std_logic_vector(7 downto 0);
    done: out std_logic
);
end timer;
architecture behavior of timer is
  signal count: std_logic_vector(7 downto 0);
  type state is range 0 to 6;
  signal current_state: state;
begin
main: process(reset, clk)
  begin
     if reset = '1' then current_state <= 0;
     elsif clk'event and clk = '1' then
      case current_state is
        when 0 => if start = '0' then current_state <= 1; end if;
                  done <= '1';
        when 1 => if start = '1' then current_state <= 2; end if;
                  done <= '1';
        when 2 => count <= time; current_state <= 3;  done <= '0';
        when 3 => if count = "00000000" then current_state <= 0;
                  else current_state <= 4; end if; done <= '0';
        when 4 => if reference = '0' then current_state <= 5; end if;
                  done <= '0';
        when 5 => if reference = '1' then current_state <= 6; end if;
                  done <= '0';
        when 6 => count <= count - '1'; current_state <= 3;
                  done <= '0';
      end case;
    end if;
  end process main;
end behavior;
```

This description is similar to a finite-state machine but it includes register operations such as $count \leq time$ (state 2) or $count \leq count - 1$ (state 6). This VHDL process includes both the operations and the control of the sequence of operations. Synthesis tools (Chap. 7) are able to translate this type of description to actual digital circuits, similar to Fig. 4.66, using library components.

4.9.2 Sequence Recognition

Consider a mobile object (it could be part of a machine tool) with two optical sensors. It moves in front of a fixed ruler with grey and white areas (Fig. 4.68). Once processed by some electronic circuitry, two bits x_1 and x_0 are generated: $x_i = 1$ if the corresponding sensor is in front of a grey area and $x_i = 0$ when the sensor is in front of a white area. As an example, in the position of Fig. 4.68, x_1x_0 is equal to 00.

If the object moves to the right (Fig. 4.69), the generated sequence is $x_1\ x_0 = \ldots$ 00 01 11 10 00 ..., and if the object moves to the left, the generated sequence is $x_1\ x_0 = \ldots$ 00 10 11 01 00 ....

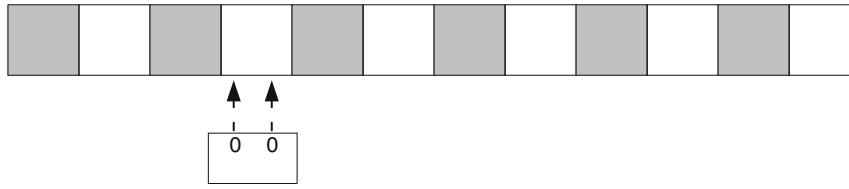


Fig. 4.68 Mobile object with two optical sensors

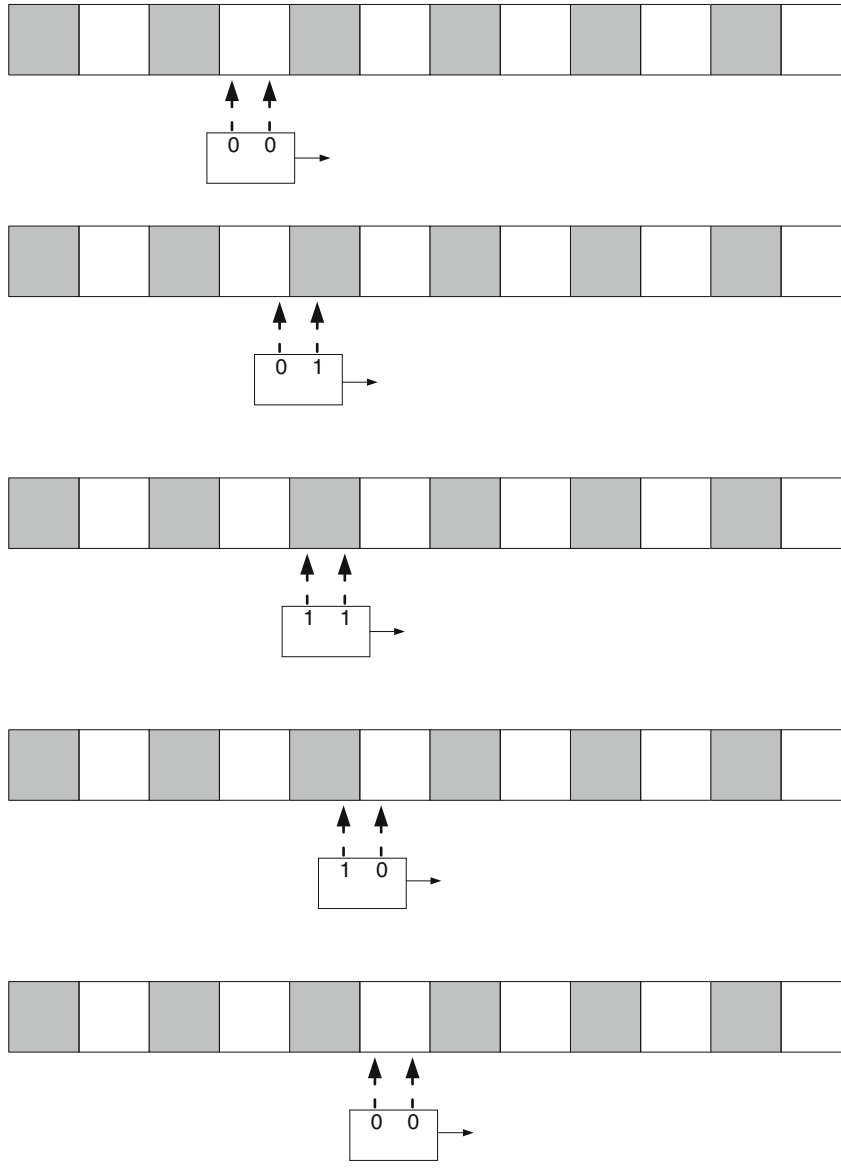


Fig. 4.69 Moving to the right

Fig. 4.70 Mealy finite-state machine that generates z

current state	$x_1\ x_0$ 00	01	10	11
A	A/0	B/0	A/1	D/1
B	B/1	B/0	A/1	C/0
C	B/1	C/1	D/0	C/0
D	A/0	C/1	D/0	D/1

next state / z

The circuit to be implemented has two binary inputs x_1 and x_0 and a binary output z equal to 0 when the object moves to the right, and equal to 1 when the object moves to the left. This specification can be implemented by a finite-state machine that detects the two sequences . . . 00 01 11 10 00 . . . and . . . 00 10 11 01 00 . . . . The Mealy finite-state machine of Fig. 4.70 has four internal states A, B, C, and D. When the input sequence is . . . 00 01 11 10 00 . . . the corresponding internal states are . . . A B C D A . . . and $z = 0$, and when the input sequence is . . . 11 01 00 10 11 . . . the corresponding internal states are . . . D C B A D . . . and $z = 1$.

The following *next_state_registered* and *output_state* processes (two-process model, Comment 4.5), in which *input_state* stands for $x_1\ x_0$, describe the finite-state machine of Fig. 4.70:

```
next_state_registered: process (reset, clk)
begin
 if reset = '1' then current_state <= A;
 elsif clk'event and clk = '1' then
  case current_state is
   when A => if input_state = "01" then current_state <= B;
             elsif input_state = "11" then current_state <= D;
             end if;
   when B => if input_state = "10" then current_state <= A;
             elsif input_state = "11" then current_state <= C;
             end if;
   when C => if input_state = "00" then current_state <= B;
             elsif input_state = "10" then current_state <= D;
             end if;
   when D => if input_state = "00" then current_state <= A;
             elsif input_state = "01" then current_state <= C;
             end if;
  end case;
 end if;
end process next_state_registered;
output_state: process (current_state, input_state)
begin
  case current_state is
   when A => z <= x1;
   when B => z <= not(x0);
   when C => z <= not(x1);
   when D => z <= x0;
  end case;
end process output_state;
```

4.10 Exercises

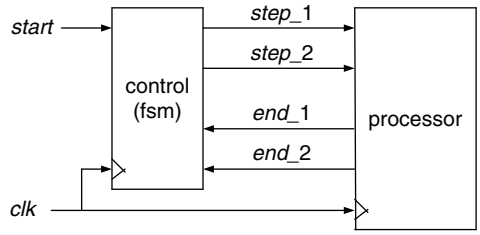
1. Synthesize with logic gates and D-type flip-flops the motor control circuit of Fig. 4.1.
2. The following program defines a Moore machine with three internal states 0, 1, and 2; two inputs x_0 and x_1; and three outputs y_0, y_1, and y_2:

```
loop
   if q = 0 then
      y0 = 0; y1 = 1; y2 = 0;
      if x0 = 1 then q = 1; else q = 2; end if;
   elsif q = 1 then
      y0 = 1; y1 = 0; y2 = 1;
      if x1 = 1 then q = 0; else q = 1; end if;
   else
      y0 = 1; y1 = 0; y2 = 0;
      if (x0 = 0) and (x1 = 0) then q = 0;
      else q = 2; end if;
   end if;
end loop;
```

   Define the corresponding state transition graph, next state table, and output table.
3. The following program defines a Mealy machine with three internal states 0, 1, and 2; two inputs x_0 and x_1; and three outputs y_0, y_1, and y_2:

```
loop
 case q is
  when 0 =>
   if x0 = 1 then q = 1; y0 = 1; y1 = 0; y2 = 1;
   else q = 2; y0 = 1; y1 = 0; y2 = 0; end if;
  when 1 =>
   if x1 = 1 then q = 0; y0 = 0; y1 = 1; y2 = 0;
   else q = 1; y0 = 1; y1 = 0; y2 = 1; end if;
  when 2 =>
   if (x0 = 0) and (x1 = 0) then q = 0; y0 = 0;
     y1 = 1; y2 = 0;
   else q = 2; y0 = 1; y1 = 0; y2 = 0; end if;
 end case;
end loop;
```

   Define the corresponding state transition graph, next state table, and output table.
4. The following system consists of a processor and a control unit. The processor executes a two-step program under the control of two binary signals *step*_1 and *step*_2, and generates two binary status signals *end*_1 and *end*_2.



The control unit is a finite-state machine whose inputs are the status signals *end*_1 and *end*_2 and an external binary command *start*, and whose outputs are the control signals *step*_1 and *step*_2. Synthesize the control unit according to the following specifications:

- On a positive edge of *start*, the control signal *step*_1 goes high and the processor starts executing the first step.
- When the first step is completed, the processor raises the status signal (flag) *end*_1.
- Then, the control signal *step*_1 goes low, the control signal *step*_2 goes high, and the processor lowers the status signal *end*_1 and starts executing the second step.
- When the second step is completed, the processor raises the status signal *end*_2.
- Then, the control signal *step*_2 goes low, the processor lowers the status signal *end*_2, and the control circuit waits for another positive edge of *start*.

5. Synthesize a bidirectional mod 60 counter.
6. Generate sequential implementations of the arithmetic circuits of Chap. 3.

References

Rabaey JM, Chandrakasan A, Nikolic B (2003) Digital integrated circuits: a design perspective. Prentice Hall, Upper Saddle River

Weste NHE, Harris DM (2010) CMOS VLSI design: a circuit and systems perspective. Pearson, Boston

Synthesis of a Processor

5

In this chapter the design of a complete digital system is presented. The system to be synthesized is a generic (general purpose) component, able to execute different algorithms. The particular algorithm is defined by the contents of a memory block that stores a (so-called) program. This type of system is generally called a processor, in this case a very simple one. It is an example of application of the synthesis methods described all along the previous chapters.

5.1 Definition

In order to introduce the concept of processor, the two examples of Chap. 1, namely the temperature controller (Example 1.1) and the chronometer (Example 1.2), are revisited.

5.1.1 Specification

The temperature controller executes Algorithm 1.1.

Algorithm 1.1 Temperature Control

```
loop
   if temp < pos - half_degree then onoff = on;
   elsif  temp > pos + half_degree then onoff = off;
   end if;
   wait for 10s;
end loop;
```

and the chronometer executes the following algorithm in which the condition *ref_positive_edge* is true on every positive edge of the signal *ref* (in VHDL this condition would be expressed as *ref'event and ref = '1'*) and *update* is a function that adds 0.1 s to the current time.

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_5

Algorithm 5.1 Chronometer

```
loop
   if reset = on then time = 0;
   elsif start = on then
      while stop = off loop
         if ref_positive_edge = true then
         time = update(time); end if;
      end loop;
   end if;
end loop;
```

Algorithm 1.1 specifies the behavior of the temperature control block of Fig. 1.1 and Algorithm 5.1 specifies the behavior of the time computation block of Fig. 1.2.

5.1.2 Design Strategy

There exist several strategies to design and to implement the digital systems specified in the preceding section. The designer will choose the best in function of parameters such as the cost, the accuracy, and the size of the final system.

A first option is to develop, for each algorithm, a completely new system that executes the corresponding algorithm but could not execute another algorithm. As a matter of fact, the design of new digital systems is the main topic of this course. However, both algorithms have some common characteristics. For example:

- Both algorithms consist of instructions that are sequentially executed; in the case of the temperature controller, the following instructions are executed at time ... t_0, t_1, t_2, t_3, ...

```
t0: if temp < pos - half_degree then onoff = on;
   elsif temp > pos + half_degree then onoff = off;
end if;
t1: wait for 10 s;
t2: if temp < pos - half_degree then onoff = on;
   elsif temp > pos + half_degree then onoff = off;
   end if;
t3: wait for 10 s;
```

- There are branches and jumps; for example, the following instruction of the temperature controller is a branch

```
if temp < pos - half_degree then onoff = on;
elsif temp > pos + half_degree then onoff = off;
```

and in the chronometer algorithm the instruction

```
while stop = off loop
    if ref_positive_edge = true then
    time = update(time); end if;
end loop;
```

implicitly includes a jump: if *stop* = *on* then the loop body is not executed.

- Both algorithms include instructions that read input values or generate output values; in the temperature controller the following instruction implies the reading of the input signals *temp* and *pos* and generates the value of the output signal *onoff*

```
if temp < pos - half_degree then onoff = on;
elsif temp > pos + half_degree then onoff = off;
```

and in the chronometer the instruction

```
while stop = off loop
    if ref_positive_edge = true then
    time = update(time); end if;
end loop;
```

reads the input signals *stop* and *ref* and generates the value of an output signal *time*.

- Some instructions execute computations; for example (temperature controller)

```
temp - pos;
```

or, in the case of the chronometer, the instruction that adds 0.1 s to the current time value

```
update(time);
```

The fact that many digital systems can be defined by algorithms that have several common characteristics suggests another design strategy. Instead of designing a specific circuit for any new system that must be implemented, a generic system that will serve to implement different systems is first developed (Fig. 5.1).

This generic system has input ports $IN_0, IN_1, IN_2, \ldots$ and output ports $OUT_0, OUT_1, OUT_2, \ldots$ that permit to read input signal values and to generate output signal values. It includes memory elements for example $X_0, X_1, X_2, \ldots$, able to store the data necessary to execute the algorithms, and it must include computing resources able to execute arithmetic and logic operations. The generic system must have the capacity to interpret instructions (commands) such as

- $X_i = A$: Store a constant A in memory element X_i.
- $X_i = IN_j$: Store the value of input port IN_j in memory element X_i.
- $OUT_i = X_j$: Transmit to output port OUT_i the value stored in memory element X_j.
- $OUT_i = A$: Transmit the constant A to output port OUT_i.
- $X_i = f(X_j, X_k)$: Compute f using as operands the data stored in X_j and X_k, and store the result in X_i, f being one of the available computation resources.

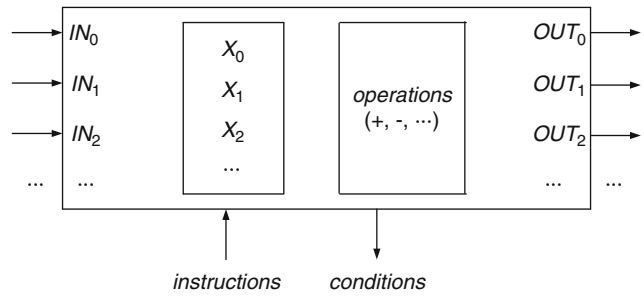


Fig. 5.1 Generic system

- *goto n*: Jump to instruction number *n*.
- *if some_condition then goto n*: Jump to instruction number *n* if some condition holds true.

Thus, this generic system receives external instructions and generates some conditions such as "X_i is equal to some constant value" or "X_i is greater than X_j."

Assume that such a generic component has been developed. It can be used to implement the temperature controller specified by the following (slightly simplified) algorithm:

Algorithm 5.2 Temperature Control (Simplified Version)

```
loop
  if temp < pos then onoff = on;
  elsif temp > pos then onoff = off;
  end if;
  wait for 10 s;
end loop;
```

The difference with Algorithm 1.1 is that the temperature accuracy is equal to 1° instead of half a degree.

A first operation is to assign ports to the input and output signals. The value *temp* of the temperature measured by the sensor is connected to input port 0. The value *pos* of the desired temperature defined by the position of a mechanical selector is connected to port 1. The value of the command *onoff* transmitted to the boiler is available at output port 0. Furthermore a time reference is necessary to measure the delay (10 s). For that an external clock is used; it generates a signal *time* (the current time expressed in seconds) that is connected to input port 2. The temperature controller can be implemented by the circuit of Fig. 5.2 under the control of a sequence of instructions (a program) belonging to a predefined set of instructions.

To complete the definition of the set of executable instructions it remains to define the available operations *f* and the jump conditions. Assume that there are two operations

$$X_i = X_j + X_k \quad \text{and} \quad X_i = X_j - X_k,$$

and two jump conditions

$$\textit{if } X_i < 0 \textit{ then goto } n \text{ and } \textit{if } X_i > 0 \textit{ then goto } n.$$

Thus there are nine instruction types (Table 5.1).

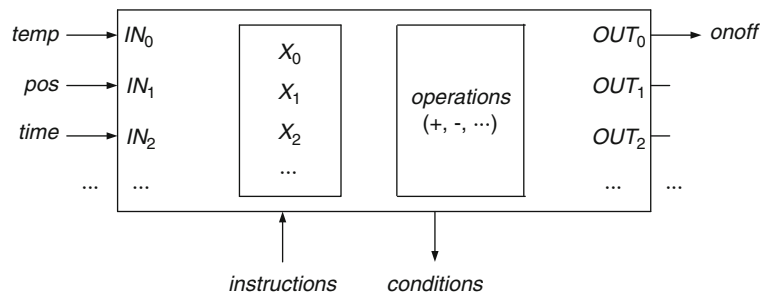


Fig. 5.2 Temperature controller

Table 5.1 Instruction types

$X_i = A$
$X_i = IN_j$
$OUT_i = X_j$
$OUT_i = A$
$X_i = X_j + X_k$
$X_i = X_j - X_k$
goto n
if $X_i < 0$ *then goto n*
if $X_i > 0$ *then goto n*

Other characteristics, for example the number of bits of the ports and of the memory elements, or the total number of ports and memory elements, will be defined later.

In order to execute the temperature control algorithm, memory elements must be assigned to the algorithm variables. For example:

X_0 stores the measured temperature read from input port 0.
X_1 stores the selector position read from input port 1.
X_2 stores the time read from the external clock.
X_3 stores the time read from the external clock at the beginning of the *wait* instruction execution.
X_4 stores the computation results.
X_5 stores the constant value 10.

The program is generated under the form of a flow diagram (Fig. 5.3).

The first instruction stores the constant value 10 in the memory element X_5. Then the value of the measured temperature is read and stored in X_0, the selector position is read and stored in X_1, and the difference $X_0 - X_1$ is computed and stored in X_4. If the difference is negative, the *onoff* output signal is set to 1 and if the difference is positive, the *onoff* output signal is set to 0. If the difference is neither negative nor positive (equal to 0) the output signal value doesn't change. It remains to execute the *wait* instruction. The current time value is read and stored in X_3. Then the current time value is read once more and stored in X_2. The difference $X_2 - X_3$ is computed and stored in X_4; it will be equal to the elapsed time since the beginning of the *wait* instruction execution. The elapsed time is compared with the constant value 10 stored in X_5. If the difference is negative, the elapsed time is smaller than 10 s and a new value of *time* is read and stored in X_2. If the difference is not negative, the elapsed time is equal to 10 s and the program execution goes back to the beginning.

To define the program under the form of a sequence of instructions, numbers must be assigned to the instructions. Except in the case of unconditional jumps (*goto n*), or in the case of conditional jumps (*if* $Xi > 0$ *then goto n* or *if* $Xi < 0$ *then goto n*) when the condition holds true, the instructions are executed in increasing order. A correct numbering of the instructions is shown in Fig. 5.3. Some unconditional jumps (instructions 6, 8, and 15) have been added in order to respect the preceding rule.

The following program, using instructions of Table 5.1, corresponds to the flow diagram of Fig. 5.3:

0: $X_5 = 10$;
1: $X_0 = IN_0$;
2: $X_1 = IN_1$;
3: $X_4 = X_0 - X_1$;
4: if $X_4 < 0$ then goto 7;
5: if $X_4 > 0$ then goto 9;
6: goto 10;

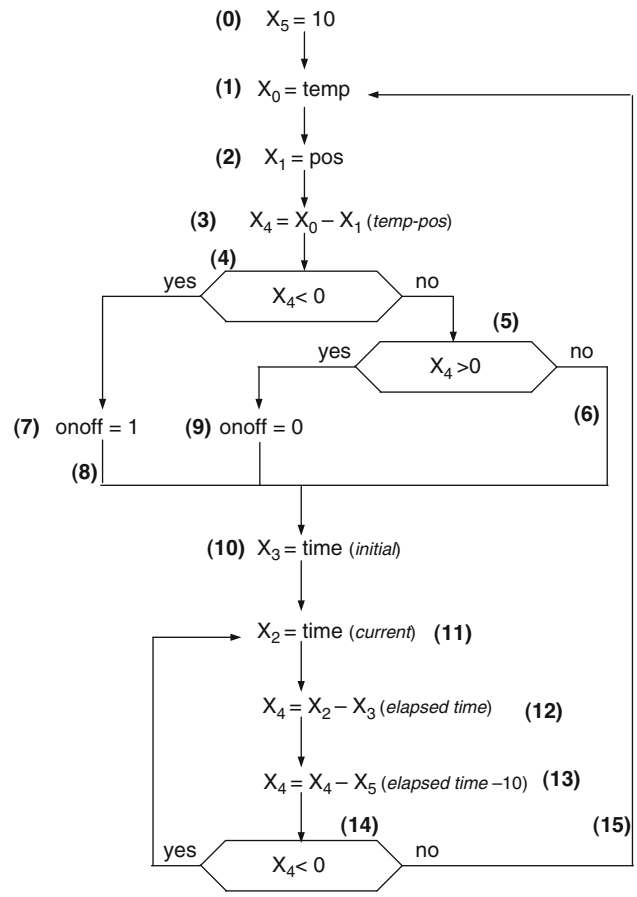


Fig. 5.3 Temperature control program

7: $OUT_0 = 1$;
8: goto 10;
9: $OUT_0 = 0$;
10: $X_3 = IN_2$;
11: $X_2 = IN_2$;
12: $X_4 = X_2 - X_3$;
13: $X_4 = X_4 - X_5$;
14: if $X_4 < 0$ then goto 11;
15: goto 1;

The generic system of Fig. 5.2 (the processor) is connected to a control unit that stores the preceding program. This control unit sequentially sends to the processor the instructions to be executed and, in the case of the conditional jump instructions, the processor sends to the control unit the result (*true* or *false*) of the jump condition evaluation.

The second example is the chronometer (Fig. 1.2) specified by Algorithm 5.1. It is assumed that the output signal *time* is expressed as a multiple of 0.1 s and that an external circuit transforms this information into a number of hours, of minutes, of seconds, and of tenths of seconds. The *update* function of Algorithm 5.1 then consists of adding one unit to the current value of signal *time*.

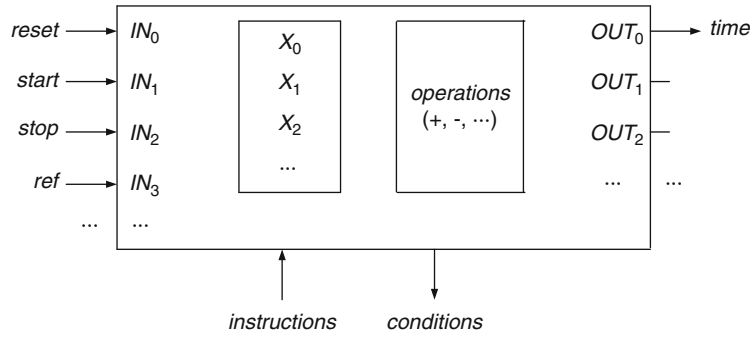


Fig. 5.4 Chronometer

The following port assignment is chosen (Fig. 5.4):

reset is connected to input port 0.
start is connected to input port 1.
stop is connected to input port 2.
ref is connected to input port 3.
time is available at output port 0.

Four memory elements are used:

X_0 stores input signal values: *reset*, *start*, or *stop*.
X_1 stores the value of *ref*.
X_2 stores the value of *time*.
X_3 stores the constant 1.

As before the program is generated under the form of a flow diagram (Fig. 5.5). The first instruction stores constant 1 in X_3. Then *reset* is read and stored in X_0. If *reset* $= 1$ then X_2 (internal value of *time*) $= 0$, X_2 is transmitted to output port 0 (*time*), and the program goes back to the beginning. If *reset* $= 0$ then *start* is read and stored in X_0. If *start* $= 0$ then the program goes back to the beginning. If *start* $= 1$ then *stop* is read and stored in X_0. If *stop* $= 1$ then the program goes back to the beginning. If *stop* $= 0$ then a positive edge of *ref* must be detected. For that, *ref* is read and stored in X_1. If *ref* $= 1$ then the program enters a loop waiting for *ref* $= 0$. When *ref* $= 0$ the program enters another loop waiting for *ref* $= 1$. Finally, the internal value of *time* is updated by adding one unit (X_3) to X_2, the updated value is transmitted to output port 0, and the program goes back to the reading of *stop*.

Comment 5.1
Several instructions are executed between the detection of two successive positive edges of *ref* (update *time*, read *stop*). Thus, the instruction execution time must be much smaller than the *ref* period (0.1 s).

The following program, using instructions of Table 5.1, corresponds to the flow diagram of Fig. 5.4:

0: $X_3 = 1$:
1: $X_0 = IN_0$;
2: if $X_0 > 0$ then goto 6;

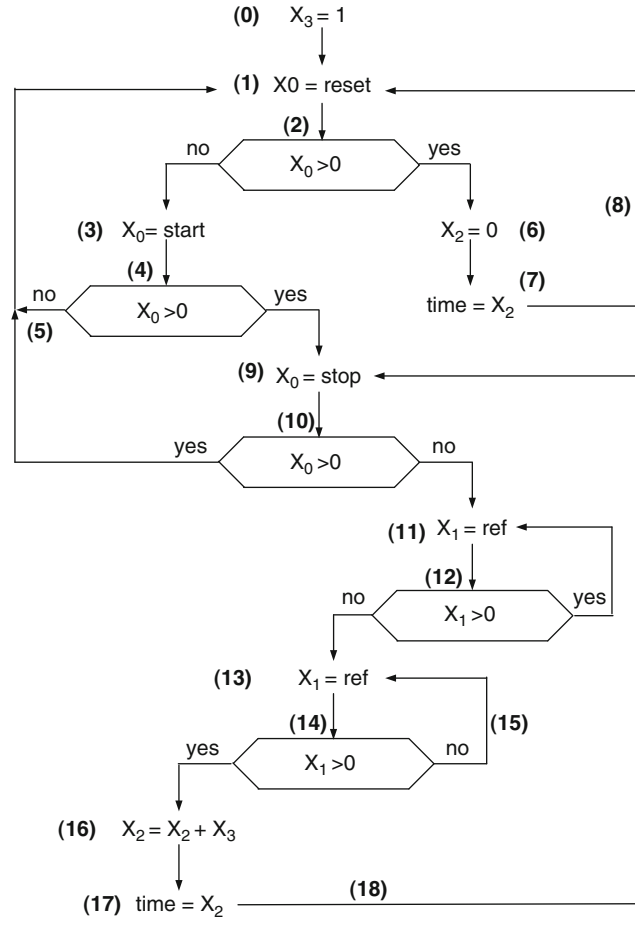


Fig. 5.5 Chronometer program

3: $X_0 = IN_1$;
4: if $X_0 > 0$ then goto 9;
5: goto 1;
6: $X_2 = 0$;
7: $OUT_0 = X_2$;
8: goto 1;
9: $X_0 = IN_2$;
10: if $X_0 > 0$ then goto 1;
11: $X_1 = IN_3$;
12: if $X_1 > 0$ then goto 11;
13: $X_1 = IN_3$;
14: if $X_1 > 0$ then goto 16;
15: goto 13;
16: $X_2 = X_2 + X_3$;
17: OUT0 $= X_2$;
18: goto 9;

5.2 Functional Specification

This section presents a formal and more precise specification of the processor.

5.2.1 Instruction Types

Each instruction type is characterized by a code and a list of parameters (Table 5.2). The code consists of one or two words that briefly describe the operation performed by the instruction.

In the case of the *OPERATION* instruction, the parameter f is an identifier of an operation (sum, difference, multiplication, and so on) that one of the processing resources can execute. Those operations will be defined later.

As an example, with this new instruction type definition, the temperature control algorithm can be rewritten under the following form:

0: ASSIGN_VALUE, 5, 10	$(X_5 = 10)$
1: DATA_INPUT, 0, 0	$(X_0 = IN_0)$
2: DATA_INPUT, 1, 1	$(X_1 = IN_1)$
3: OPERATION, 0, 1, 4, subtract	$(X_4 = X_0 - X_1)$
4: JUMP_NEG, 4, 7	$(if\ X_4 < 0\ then\ goto\ 7)$
5: JUMP_POS, 4, 9	$(if\ X_4 > 0\ then\ goto\ 9)$
6: JUMP, 10	$(goto\ 10)$
7: OUTPUT_VALUE, 0, 1	$(OUT_0 = 1)$
8: JUMP, 10	$(goto\ 10)$
9: OUTPUT_VALUE, 0, 0	$(OUT_0 = 0)$
10: DATA_INPUT, 3, 2	$(X_3 = IN_2)$
11: DATA_INPUT, 2, 2	$(X_2 = IN_2)$
12: OPERATION, 2, 3, 4, subtract	$(X_4 = X_2 - X_3)$
13: OPERATION, 4, 5, 4, subtract	$(X_4 = X_4 - X_5)$
14: JUMP_NEG, 4, 11	$(if\ X_4 < 0\ then\ goto\ 11)$
15: JUMP, 1	$(goto\ 1)$

5.2.2 Specification

First, the processor behavior is slightly modified. In the previous sections it was assumed that the processor receives instructions from an external circuit (or an external programmer) and that this external circuit (or programmer) receives information about the value of some data stored within the processor, under the form of conditions such as "X_i is positive" or "X_k is negative." Then, the external circuit (or programmer) takes a decision about the next instruction to be executed in function of those conditions.

The new specification assumes that the next instruction number is internally computed by the processor itself. The modified structure is shown in Fig. 5.6: the program is stored in a memory whose address *number* is generated by the processor. Assume that the current value of *number* is n. Then, the processor reads the instruction stored at address n, it executes the instruction, and it computes the next

Table 5.2 Instructions: code, parameters, and corresponding operation

Code and list of parameters	Operation
ASSIGN_VALUE, k, A	$X_k = A$
DATA_INPUT, k, j	$X_k = IN_j$
DATA_OUTPUT, i, j	$OUT_i = X_j$
OUTPUT_VALUE, i, A	$OUT_i = A$
OPERATION, i, j, k, f	$X_k = f(X_i, X_j)$
JUMP, N	*goto* N
JUMP_POS, i, N	*if* $Xi > 0$ *goto* N
JUMP_NEG, i, N	*if* $Xi < 0$ *goto* N

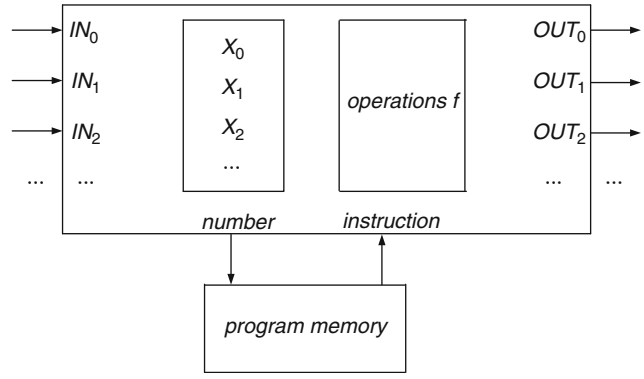


Fig. 5.6 Stored program architecture

instruction address. The latter generally is $n + 1$ except in the case of jumps, in which case the next instruction address is one of the instruction parameters. This is the so-called von Neumann architecture (stored program architecture).

Define vectors $IN = (IN_0, IN_1, IN_2, \dots)$, $OUT = (OUT_0, OUT_1, OUT_2, \dots)$ and $X = (X_0, X_1, X_2, ,,,)$ whose components are m-bit vectors. The number of components and the value of m will be defined later. The following algorithm defines the processor behavior:

Algorithm 5.3 Processor specification

```
number = 0;
loop
   case instruction is
      when (ASSIGN_VALUE, k, A) =>
         X(k) = A; number = number + 1;
      when (DATA_INPUT, k, j) =>
         X(k) = IN(j); number = number + 1;
      when (DATA_OUTPUT, i, j) =>
         OUT(i) = X(j); number = number + 1;
      when (OUTPUT_VALUE, i, A) =>
         OUT(i) = A; number = number + 1;
      when (OPERATION, i, j, k, f) =>
         X(k) = f(X(i), X(j)); number = number + 1;
      when (JUMP, N) =>
         number = N;
      when (JUMP_POS, i, N) =>
         if X(i) > 0 then number = N;
         else number = number + 1; end if;
      when (JUMP_NEG, i, N) =>
         if X(i) < 0 then number = N;
         else number = number + 1; end if;
   end case;
end loop;
```

Table 5.3 Number of instructions

Code and list of parameters	Operations	Number of instructions
ASSIGN_VALUE, *k*, *A*	$X_k = A$	$16 \cdot 256 = 4096$
DATA_INPUT, *k*, *j*	$X_k = IN_j$	$16 \cdot 8 = 128$
DATA_OUTPUT, *i*, *j*	$OUT_i = X_j$	$8 \cdot 16 = 128$
OUTPUT_VALUE, *i*, *A*	$OUT_i = A$	$8 \cdot 256 = 2048$
OPERATION, *i*, *j*, *k*, *f*	$X_k = f(X_i, X_j)$	$16 \cdot 2 \cdot 16 \cdot 16 = 8192$
JUMP, *N*	*goto N*	256
JUMP_POS, *i*, *N*	*if Xi* > 0 *goto N*	$16 \cdot 256 = 4096$
JUMP_NEG, *i*, *N*	*if Xi* < 0 *goto N*	$16 \cdot 256 = 4096$

Comment 5.2
Table 5.2 defines eight instruction types. The number of different instructions depends on the parameter sizes. Assume that the internal memory X stores sixteen 8-bit data and that the program memory stores at most 256 instructions so that the addresses are also 8-bit vectors. Assume also that there are two different operations f. The number of instructions of each type is shown in Table 5.3: there are 16 memory elements X_i, 256 constants A, 8 input ports IN_i, 8 output ports OUT_i, 256 addresses N, and 2 operations f.

The total number is 4096 + 128 + 128 + 2048 + 8192 + 256 + 4096 + 4096 = 23,040, a number greater than 2^{14}. Thus, the minimum number of bits needed to associate a different binary code to every instruction is 15.

5.3 Structural Specification

The implementation method is top-down. The first step was the definition of a functional specification (Sect. 5.2). Now, this specification will be translated to a block diagram.

5.3.1 Block Diagram

To deduce a block diagram from the functional specification (Algorithm 5.3) the following method is used: extract from the algorithm the set of processed data, the list of data transfers, and the list of data operations.

The processed data are the following:

- Input data: There are eight input ports IN_i and an input signal *instruction*.
- Output data: There are eight output ports OUT_i and an output signal *number*.
- Internal data: There are 16 internally stored data X_i.

The data transfers are the following:

- Transmit the value of a memory element X_j or of a constant A to an output port.
- Update *number* with *number* + 1 or with a jump address N.
- Store in a memory element X_k a constant A, an input port value IN_j, or the result of an operation f.

The operations are $\{f(X_i, X_j)\}$ with all possible functions f.
The proposed block diagram is shown in Fig. 5.7. It consists of five components.

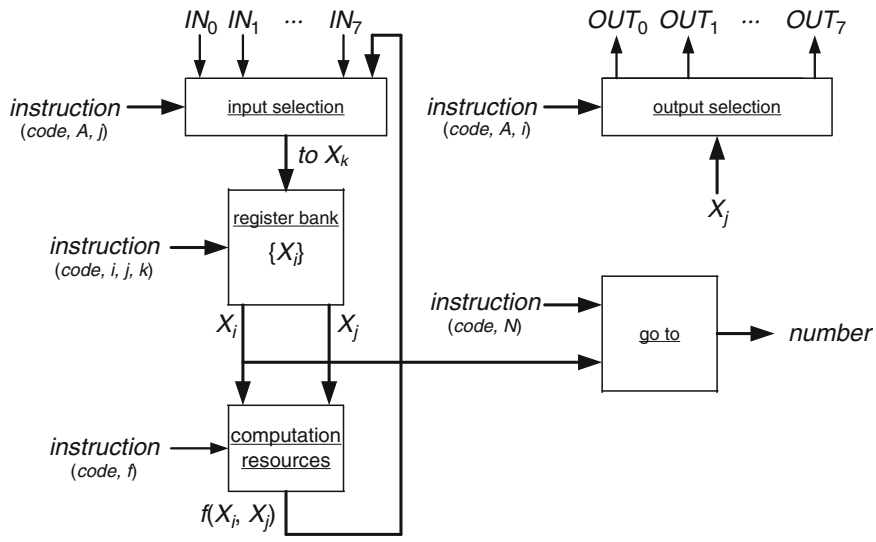


Fig. 5.7 Block diagram

- Register bank: This component contains the set of internal memory elements $\{X_i\}$. It is a 16-word memory with a data input *to* X_k and two data outputs X_i and X_j. It will be implemented in such a way that within a clock period two data X_i and X_j can be read and an internal memory element X_k can be updated (written). Thus the operation $X_k = f(X_i, X_j)$ is executed in one clock cycle. This block is controlled by the instruction code and by the parameters i, j, and k.
- Output selection: This component transmits to the output port OUT_i the rightmost register bank output X_j or a constant A. It is controlled by the instruction code and by the parameters i and A.
- Go to: It is a programmable counter: it stores the current value of *number*; during the execution of each instruction, it replaces *number* by *number* + 1 or by a jump address N. It is controlled by the instruction code, by the parameter N, and by the leftmost register bank output X_i whose most significant bit value (sign bit) is used in the case of conditional jumps.
- Input selection: This component selects the data to be sent to the register bank, that is, an input port IN_j, a constant A, or the result of an operation f. It is controlled by the instruction code and by the parameters j and A.
- Computation resources: This is an arithmetic unit that computes a function f with operands that are the two register bank outputs X_i and X_j. It is controlled by the instruction code and by the parameter f.

The set of instruction types has already been defined (Table 5.2). All input data (IN_j), output data (OUT_j), and internally stored data (X_i, X_j, X_k) are 8-bit vectors. It remains to define the size of the instruction parameters and the arithmetic operations:

- There are eight input ports and eight output ports and the register bank stores 16 words; thus i, j, and k are 4-bit vectors.
- The maximum number of instructions is 256 so that *number* is an 8-bit natural.

With regard to the arithmetic operations, the sixteen 8-bit vectors X_0 to X_{15} are interpreted as 2's complement integers ((3.4) with $n = 8$). Thus $-128 \leq X_i \leq 127$, $\forall i = 0$–15. There are two operations f: $X_k = (X_i + X_j) \bmod 256$ and $X_k = (X_i - X_j) \bmod 256$.

The instruction encoding will be defined later.

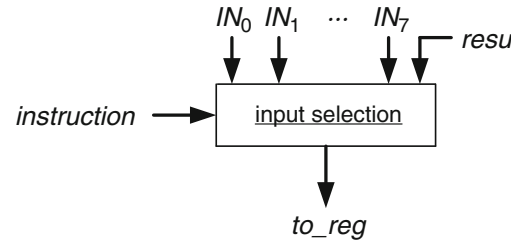


Fig. 5.8 Input selection

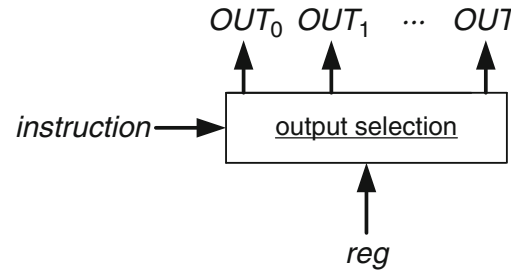


Fig. 5.9 Output selection

5.3.2 Component Specification

Each component will be functionally described.

5.3.2.1 Input Selection

To define the working of the input selection component (Fig. 5.8), extract from Algorithm 5.3 the instructions that select the data inputted to the register bank:

Algorithm 5.4 Input Selection

```
loop
   case instruction is
      when (ASSIGN_VALUE, k, A) => to_reg = A;
      when (DATA_INPUT, k, j) => to_reg = IN(j);
      when (OPERATION, i, j, k, f) => to_reg = result;
      when others => to_reg = don't care;
   end case;
end loop;
```

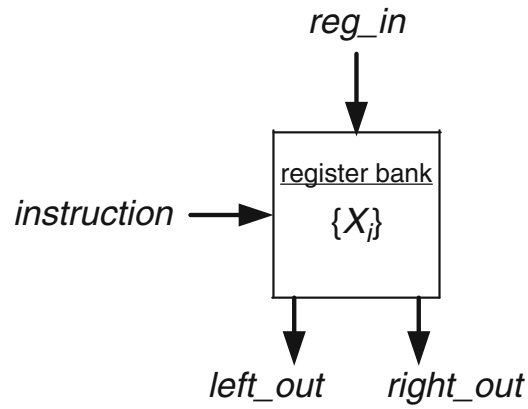
5.3.2.2 Output Selection

To define the working of the output selection component (Fig. 5.9) extract from Algorithm 5.3 the instructions that select the data outputted to the output ports:

Algorithm 5.5 Output Selection

```
loop
   case instruction is
     when (DATA_OUTPUT, i, j) =>
        OUT(i) = reg;
```

Fig. 5.10 Register bank




```
      when (OUTPUT_VALUE, i, A) =>
         OUT(i) = A;
   end case;
end loop;
```

The output ports are registered outputs: if the executed instruction is neither *DATA_OUTPUT* nor *OUTPUT_VALUE*, or if $k \neq i$, the value of OUT_k does not change.

5.3.2.3 Register Bank

The register bank is a memory that stores sixteen 8-bit words (Fig. 5.10). Its working is described by the set of instructions of Algorithm 5.3 that read or write some memory elements (X_i, X_j, X_k). It is important to observe (Fig. 5.7) that in the case of the *DATA_OUTPUT* instruction X_j is the rightmost output of the register bank and in the case of the *JUMP_POS* and *JUMP_NEG* instructions X_i is the leftmost output of the register bank.

Algorithm 5.6 Register Bank

```
loop
   case instruction is
      when (ASSIGN_VALUE, k, A) =>
        X(k) = reg_in;
        left_out = don't care; right_out = don't care;
      when (DATA_INPUT, k, j) =>
        X(k) = reg_in;
        left_out = don't care; right_out = don't care;
      when (DATA_OUTPUT, i, j) =>
        right_out = X(j);
        left_out = don't care;
      when (OPERATION, i, j, k, f) =>
        X(k) = reg_in;
        left_out = X(i); right_out = X(j);
      when (JUMP_POS, i, N) =>
        left_out = X(i);
        right_out = don't care;
      when (JUMP_NEG, i, N) =>
        left_out = X(i);
        right_out = don't care;
```

```
        when others =>
          left_out = don't care; right_out = don't care;
    end case;
end loop;
```

5.3.2.4 Computation Resources

To define the working of the computation resources component (Fig. 5.11) extract from Algorithm 5.3 the instruction that computes *f*. Remember that there are only two operations: *addition* and *difference*.

Algorithm 5.7 Computation Resources

```
loop
    case instruction is
        when (OPERATION, i, j, k, f) =>
          if f = addition then
            result = (left_in + right_in) mod 256;
        else
           result = (left_in - right_in) mod 256;
        end if;
        when others =>
          result = don't care;
    end case;
end loop;
```

5.3.2.5 Go To

This component (Fig. 5.12) is in charge of computing the address of the next instruction within the program memory:

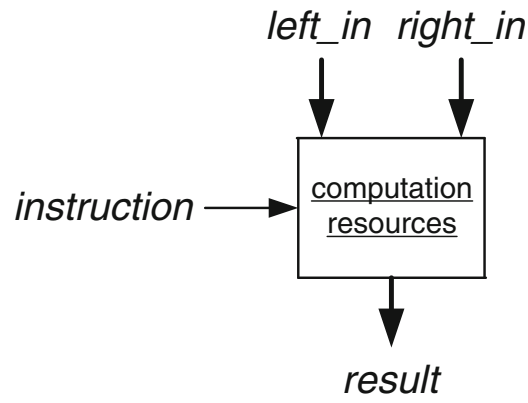


Fig. 5.11 Computation resources

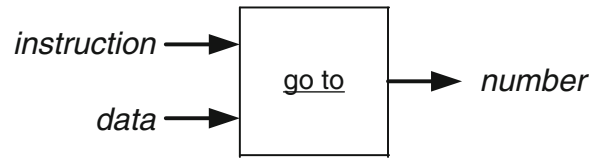


Fig. 5.12 Go to component

Algorithm 5.8 Go To

```
number = 0;
loop
   case instruction is
      when (JUMP, N) =>
         number = N;
      when (JUMP_POS, i, N) =>
         if data > 0 then number = N;
         else number = number + 1; end if;
      when (JUMP_NEG, i, N) =>
         if data < 0 then number = N;
         else number = number + 1; end if;
      when others => number = number + 1;
   end case;
end loop;
```

5.4 Component Implementation

The final step of this top-down implementation is the synthesis of all components that have been functionally defined in Sect. 5.3.2. Every component is implemented with logic gates, multiplexers, flip flops, and so on. A VHDL model of each component will also be generated.

5.4.1 Input Selection Component

The input and output signals of this component are shown in Fig. 5.8 and its functional specification is defined by Algorithm 5.4. It is a combinational circuit: the value of *to_reg* only depends on the current value of inputs *instruction*, IN_0 to IN_7, and *result*.

Instead of inputting the complete *instruction* code to the component, a 2-bit *input_control* variable (Table 5.4) that classifies the instruction types into four categories, namely *ASSIGN_VALUE*, *DATA_INPUT*, *OPERATION*, and *others*, is defined. Once the encoding of the instructions will be defined, an instruction decoder that generates (among others) this 2-bit variable will be designed.

From Algorithm 5.4 and Table 5.4 the following description is obtained:

Algorithm 5.9 Input Selection Component

```
loop
    case input_control is
       when 00 => to_reg = A;
       when 01 => to_reg = IN(j);
       when 10 => to_reg = result;
```

Table 5.4 Encoded instruction types (input instructions)

Instruction type	*input_control*
ASSIGN_VALUE	00
DATA_INPUT	01
OPERATION	10
Others	11

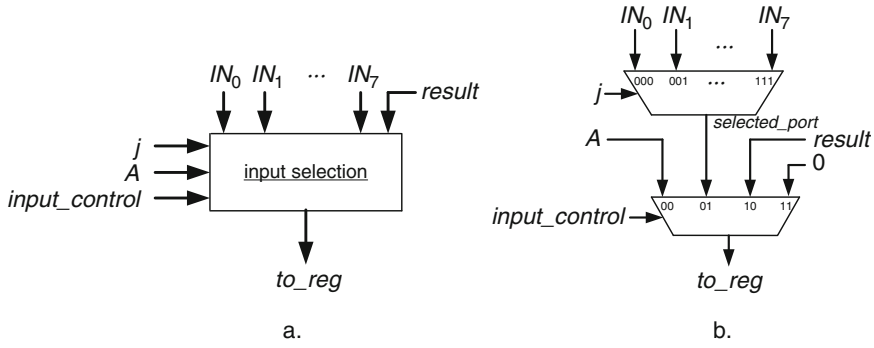


Fig. 5.13 Input selection implementation

```
        when 11 => to_reg = don't care;
   end case;
end loop;
```

The component inputs are *input_control*, *A*, *j* (parameters included in the instruction), IN_0 to IN_7, and *result*, and the component output is *to_reg* (Fig. 5.13a). A straightforward implementation with two multiplexers is shown in Fig. 5.13b.

The following VHDL model describes the circuit of Fig. 5.13b. Its architecture consists of two processes that describe the 8-bit MUX8-1 and MUX4-1:

```
package main_parameters is
   constant m: natural := 8; -- m-bit processor
end main_parameters;

library IEEE; use IEEE.std_logic_1164.all;
use work.main_parameters.all;
entity input_selection is
port (
   IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7:
      in std_logic_vector(m-1 downto 0);
   A, result: in std_logic_vector(m-1 downto 0);
   j: in std_logic_vector(2 downto 0);
   input_control: in std_logic_vector(1 downto 0);
   to_reg: out std_logic_vector(m-1 downto 0)
);
end input_selection;

architecture structure of input_selection is
   signal selected_port: std_logic_vector(m-1 downto 0);
begin
   first_mux: process(j,IN0,IN1,IN2,IN3,IN4,IN5,IN6,IN7)
    begin
        case j is
            when "000" => selected_port <= IN0;
            when "001" => selected_port <= IN1;
            ...
            when "110" => selected_port <= IN6;
```

```
            when others => selected_port <= IN7;
        end case;
    end process;
  second_mux: process(input_control,A,selected_port,result)
  begin
      case input_control is
        when "00" => to_reg <= A;
        when "01" => to_reg <= selected_port;
        when "10" => to_reg <= result;
        when others => to_reg <= (others => '0');
      end case;
  end process;
end structure;
```

5.4.2 Computation Resources

The functional specification of the computation resources component is defined by Algorithm 5.7. In fact, the working of this component when the executed instruction is not an operation (*others* in Algorithm 5.7) doesn't matter. As before, instead of inputting the complete *instruction* to the component, a control variable f equal to 0 in the case of an addition and to 1 in the case of a subtraction will be generated by the instruction decoder. This is the component specification:

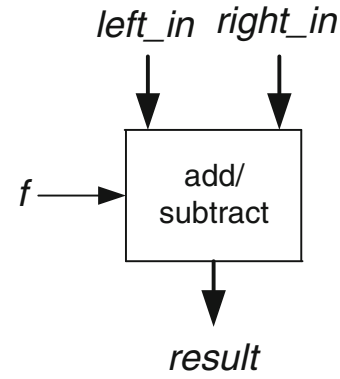
Algorithm 5.10 Arithmetic Unit

```
if f = 0 then result = (left_in + right_in) mod 256;
else result = (left_in - right_in) mod 256;
end if;
```

This component (Fig. 5.14) is a mod 256 adder/subtractor controlled by a control input f (Fig. 3.3 with $n = 8$ and $a/s = f$ and without the output *ovf*).

The following VHDL model uses the IEEE arithmetic packages. All commercial synthesis tools use those packages and would generate an efficient arithmetic unit. The package *main_parameters* has already been defined before (Sect. 5.4.1):

Fig. 5.14 Arithmetic unit



```
library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.main_parameters.all;
entity computation_resources is
port (
   left_in, right_in: in std_logic_vector(m-1 downto 0);
   f: in std_logic;
   result: out std_logic_vector(m-1 downto 0)
);
end computation_resources;architecture behavior of computation_resources is
begin
   process(f, left_in, right_in)
   begin
        if f = '0' then result <= left_in + right_in;
        else result <= left_in - right_in;
        end if;
   end process;
end behavior;
```

5.4.3 Output Selection

The input and output signals are shown in Fig. 5.9 and its functional specification is defined by Algorithm 5.5. It is a sequential circuit: the outputs OUT_0 to OUT_7 are registered. As before, instead of inputting the complete *instruction* code to the component, two binary control variables *out_en* and *out_sel* are defined (Table 5.5); they will be generated by the instruction decoder.

From Algorithm 5.5 and Table 5.5 the following description is obtained:

Algorithm 5.11 Output Selection Component

```
loop
    case (out_en, out_sel) is
       when 11 => OUT(i) = reg;
       when 10 => OUT(i) = A;
       when others => null;
    end case;
end loop;
```

The component inputs are *out_en*, *out_sel*, *A*, *i* (parameters included in the instruction), and *reg*, and the component outputs are OUT_0 to OUT_7 (Fig. 5.15a). An implementation is shown in Fig. 5.15b. The outputs OUT_0 to OUT_7 are stored in eight registers, each of them with a *CEN* (clock enable) control input. The clock signal is not represented in Fig. 5.15b. An address decoder and

Table 5.5 Encoded instruction types (output instructions)

Instruction type	*out_en*	*out_sel*
DATA_OUTPUT	1	1
OUTPUT_VALUE	1	0
Others	0	–

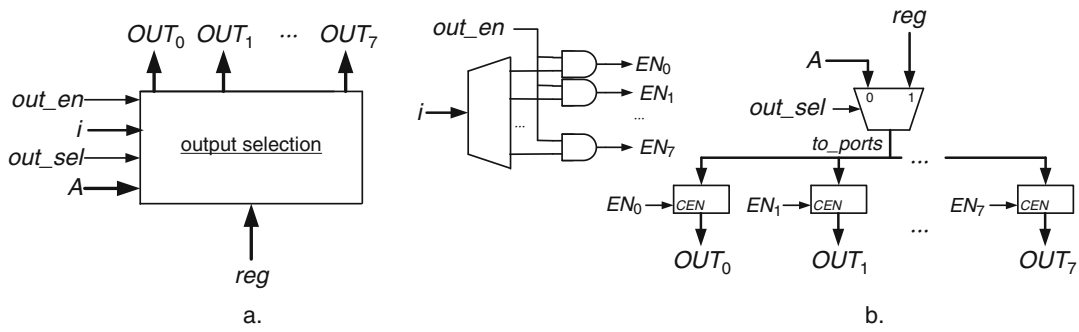


Fig. 5.15 Output selection implementation

a set of AND2 gates generate eight signals EN_0 to EN_7 that enable the clock signals of the output registers. Thus, the clock of the output register number s is enabled if $s = i$ and $out\_en = 1$. The value that is stored in the selected register is either A (if $out\_sel = 0$) or reg (if $out\_sel = 1$). If $out\_en = 0$ none of the clock signals is enabled so that the value of out_sel doesn't matter (Table 5.5).

The following VHDL model describes the circuit of Fig. 5.15b. Its architecture consists of four processes that describe the 3-to-8 address decoder, the set of eight AND2 gates, the 8-bit MUX2-1, and the set of output registers. The eight outputs of the address decoder are defined as vector *DEC_OUT*:

```
library ieee; use ieee.std_logic_1164.all;
use work.main_parameters.all;
entity output_selection is
port (
   A, reg: in std_logic_vector(m-1 downto 0);
   clk, out_en, out_sel: in std_logic;
   i: in std_logic_vector(2 downto 0);
   OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7:
     out std_logic_vector(m-1 downto 0)
);
end output_selection;

architecture structure of output_selection is
   signal EN: std_logic_vector(0 to 7);
   signal DEC_OUT: std_logic_vector(0 to 7);
   signal to_ports: std_logic_vector(m-1 downto 0);
begin
   decoder: process(i)
   begin
     case i is
         when "000" => DEC_OUT <= "10000000";
         when "001" => DEC_OUT <= "01000000";
         when "010" => DEC_OUT <= "00100000";
         .....
         when "110" => DEC_OUT <= "00000010";
         when others => DEC_OUT <= "00000001";
     end case;
   end process;
```

```
  and_gate: process(DEC_OUT, out_en)
  begin
       for i in 0 to 7 loop EN(i) <= DEC_OUT(i) AND out_en;
       end loop;
  end process;
  multiplexer: process(out_sel, A, reg)
  begin
       if out_sel = '0' then to_ports <= A;
       else to_ports <= reg; end if;
  end process;
  output_registers: process(clk)
  begin
     if clk'event and clk = '1' then
        case EN is
            when "10000000" => OUT0 <= to_ports;
            when "01000000" => OUT1 <= to_ports;
            ..........
            when "00000001" => OUT7 <= to_ports;
            when others => null;
        end case;
     end if;
  end process;
end structure;
```

5.4.4 Register Bank

The input and output signals are shown in Fig. 5.10 and its functional specification is defined by Algorithm 5.6. A control variable *write_reg* is defined (Table 5.6); it will be generated by the instruction decoder.

From Algorithm 5.6 and Table 5.6 the following description is obtained. It takes into account the fact that in all instructions *left_out* $= X_i$ or *don't care* and *right_out* $= X_j$ or *don't care*.

Algorithm 5.12 Register Bank Component

```
loop
  if write_reg = 1 then X(k) = reg_in; end if;
  right_out = X(j); left_out = X(i);
end loop;
```

The component inputs are *write_reg*, *i*, *j*, *k* (parameters included in the instruction), and *reg_in*, and the component outputs are *left_out* and *right_out* (Fig. 5.16a). An implementation is shown in Fig. 5.16b. The bank is constituted of 16 registers X_0 to X_{15}, each of them with a *CEN* (clock enable)

Table 5.6 Encoded instruction types (memory instructions)

Instruction type	*write_reg*
ASSIGN_VALUE	1
DATA_INPUT	1
OPERATION	1
Others	0

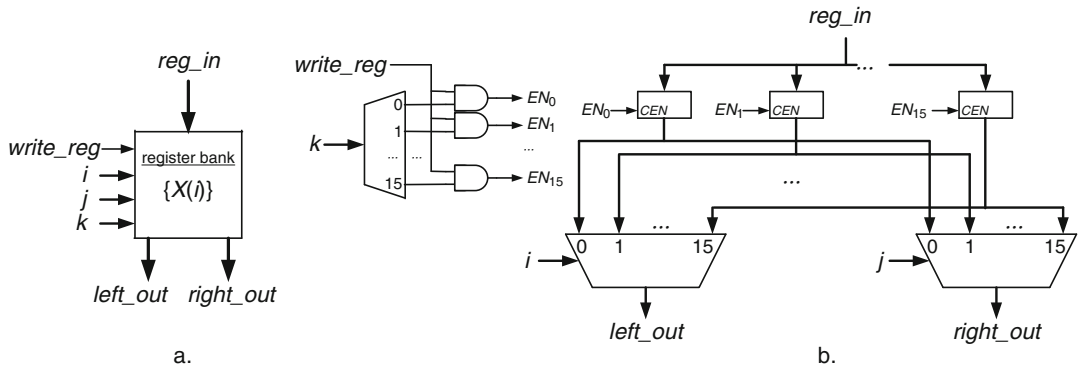


Fig. 5.16 Register bank implementation

control input. The clock signal is not represented in Fig. 5.16b. An address decoder and a set of AND2 gates generate 16 signals EN_0 to EN_{15} that enable the clock signals of the registers. Thus, the clock of register number s is enabled if $s = k$ and $write\_reg = 1$. The value that is stored in the selected register is *reg_in*. If $write\_reg = 0$ none of the clock signals is enabled. The outputs *left_out* and *right_out* are generated by two 8-bit MUX16-1 controlled by i and j.

The following VHDL model describes the circuit of Fig. 5.16b. Its architecture consists of four processes that describe the address decoder along with the set of AND2 gates, the set of registers, the left 8-bit MUX16-1, and the right 8-bit MUX16-1. The decoder model uses a conversion function *conv_integer* included in the IEEE arithmetic package:

```
library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.main_parameters.all;
entity register_bank is
port (
   reg_in: in std_logic_vector(m-1 downto 0);
   clk, write_reg: in std_logic;
   i, j, k: in std_logic_vector(3 downto 0);
   left_out, right_out:
       out std_logic_vector(m-1 downto 0)
);
end register_bank;
architecture structure of register_bank is
   type memory is array (0 to 15) of
       std_logic_vector(m-1 downto 0);
   signal X: memory;
   signal EN: std_logic_vector(0 to 15);
begin
   decoder: process(k, write_reg)
   begin
     for i in 0 to 15 loop
         if i < conv_integer(k) then EN(i) <= '0';
         elsif i = conv_integer(k) then EN(i) <= write_reg;
```

```
        else EN(i) <= '0'; end if;
        end loop;
     end process;
bank_registers: process(clk)
    begin
       if clk'event and clk = '1' then
         for i in 0 to 15 loop
             if EN(i) = '1' then X(i) <= reg_in;
             end if;
         end loop;
       end if;
    end process;
    first_multiplexer: process(i, X)
    begin
        case i is
          when "0000" => left_out <= X(0);
          ......
          when "1110" => left_out <= X(14);
          when others => left_out <= X(15);
        end case;
    end process;
    second_multiplexer: process(j, X)
    begin
        case j is
          when "0000" => right_out <= X(0);
          ..........
          when "1110" => right_out <= X(14);
          when others => right_out <= X(15);
        end case;
    end process;
end structure;
```

Comment 5.3

Instead of implementing the register bank with components such as decoders, multiplexers, registers, and gates, a better option is to use a memory macrocell (Chap. 7). The following functional description of the register bank would be translated by most commercial synthesis tools to a memory block that implements the register bank behavior:

```
architecture behavior of register_bank is
   type memory is
     array (0 to 15) of std_logic_vector(m-1 downto 0);
   signal X: memory;
   signal ii, jj, kk: integer range 0 to 15;
begin
   ii <= conv_integer(i); jj <= conv_integer(j);
   kk <= conv_integer(k);
   bank_registers: process(clk)
   begin
      if clk'event and clk = '1' then
```

Table 5.7 Encoded instruction types (jump instructions)

Instruction type	*num_sel*
JUMP	1110
JUMP_POS	1100
JUMP_NEG	1101
Others	00--, 01-- or 10--

```
        if write_reg = '1' then
        X(conv_integer(k)) <= reg_in; end if;
       end if;
   end process;
     left_out <= X(conv_integer(i));
     right_out <= X(conv_integer(j));
end behavior;
```

5.4.5 Go To Component

The input and output signals are shown in Fig. 5.12 and its functional specification is defined by Algorithm 5.8. A 4-bit control variable *numb_sel* is defined (Table 5.7); it will be generated by the instruction decoder.

From Algorithm 5.8 and Table 5.7 the following description is obtained. A *reset* input has been added: it defines the initial value of the output signal *number*.

Algorithm 5.13 Go To Component

```
if reset = 1 then number = 0;
loop
  case num_sel is
    when 1110 => number = N;
    when 1100 =>
      if data > 0 then number = N;
      else number = number + 1; end if;
    when 1101 =>
      if data < 0 then number = N;
      else number = number + 1; end if;
    when others => number = number + 1;
  end case;
end loop;
```

The component inputs are *number_sel*, *data*, *reset*, and *N* (parameter included in the instruction), and the component output is *number* (Fig. 5.17a). An implementation is shown in Fig. 5.17b. The *go to* component is made up of a programmable counter controlled by a *load* signal and two combinational circuits (Table 5.8): the first (*sign*) generates $pos = 1$, $neg = 0$ if $data > 0$, $pos = 0$, $neg = 1$ if data < 0, and $pos = neg = 0$ if $data = 0$; the second (*load*) generates $load = 1$ if *num_sel* $= 1110$ (*JUMP*) or if *numb_sel* $= 1100$ and $pos = 1$ (*JUMP_POS*) or when *numb_sel* $= 1101$ and $neg = 1$ (*JUMP_NEG*).

The following VHDL model describes the circuit of Fig. 5.17b. Its architecture consists of three processes that describe the combinational circuit *sign*, the combinational circuit *load*, and the programmable counter. The numbers are represented in 2's complement so that the sign of *data* is its most significant bit $data_7$. Hence, $data < 0$ if $data_7 = 1$, and $data > 0$ if $data_7 = 0$ and $data \neq 0$.

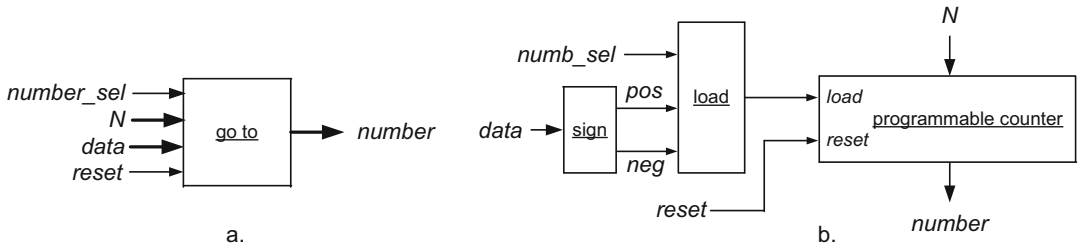


Fig. 5.17 Go to component implementation

Table 5.8 Combinational circuits *sign* and *load*

Data	*pos*	*neg*	*numb_sel*	*load*
>0	1	0	1110	1
=0	0	0	1100	pos
<0	0	1	1101	neg
			Others	0

The user-defined package *main_parameter* must contain the definition of a constant *zero*:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
package main_parameters is
   constant m: natural := 8; -- m-bit processor
   constant zero: std_logic_vector(m-1 downto 0) :=
      conv_std_logic_vector(0, m);
end main_parameters;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.main_parameters.all;
entity go_to is
port (
   N, data: in std_logic_vector(m-1 downto 0);
   clk, reset: in std_logic;
   numb_sel: in std_logic_vector(3 downto 0);
   number: inout std_logic_vector(m-1 downto 0)
);
end go_to;
architecture structure of go_to is
   signal pos, neg, load: std_logic;
begin
   sign_computation: process(data)
   begin
        if data(m-1) = '1' then pos <= '0'; neg <= '1';
        elsif data = zero then pos <= '0'; neg <= '0';
        else pos <= '1'; neg <= '0'; end if;
   end process;
   load_condition: process(numb_sel, pos, neg)
```

```
  begin
      case numb_sel is
         when "1110" => load <= '1';
         when "1100" => load <= pos;
         when "1101" => load <= neg;
         when others => load <= '0';
      end case;
  end process;
  programmable_counter: process(clk, reset)
  begin
     if reset = '1' then number <= zero;
     elsif clk'event and clk = '1' then
        if load = '1' then number <= N;
        else number <= number + 1; end if;
     end if;
  end process;
end structure;
```

5.5 Complete Processor

The five components that will be used to synthesize the processor have been implemented and synthesizable VHDL models have been generated. To complete the design, it remains to assemble the five components and to add the instruction decoder that generates some of the control signals. For that the encoding of the instructions must be defined.

5.5.1 Instruction Encoding

The instructions defined above consist of names such as *ASSIGN_VALUE* and *DATA_INPUT*, and parameters such as an index i, a constant A, and an address N. To implement the circuit those names must be encoded under the form of binary vectors.

The encoding of Table 5.9 is used. The instruction name *OPERATION* has been split into two instruction names *OPERATION_ADD* and *OPERATION_SUB*. Four bits $C_{15..12}$ encode the instruction name. Bits $C_{15..13}$ define the instruction type and C_{12} defines the particular operation (add or subtract) in the case of an operation and defines the condition (positive or negative) in the case of a conditional jump. The other 12 bits $C_{11..0}$ are the instruction parameters: an 8-bit number A, an 8-bit address N, and 3- or 4-bit indexes i, j, and k.

Table 5.9 Instruction encoding

	$C_{15..12}$	$C_{11..8}$	$C_{7..4}$	$C_{3..0}$	Operation
ASSIGN_VALUE	0000	$A_{7..4}$	$A_{3..0}$	k	$X(k) = A$
DATA_INPUT	0010	–	j	k	$X(k) = IN(j)$
DATA_OUTPUT	1010	i	j	-	$OUT(i) = X(j)$
OUTPUT_VALUE	1000	i	$A_{7..4}$	$A_{3..0}$	$OUT(i) = A$
OPERATION_ADD	0100	i	j	k	$X(k) = X(i) + X(j)$
OPERATION_SUB	0101	i	j	k	$X(k) = X(i) - X(j)$
JUMP	1110	–	$N_{7..4}$	$N_{3..0}$	go to N
JUMP_POS	1100	i	$N_{7..4}$	$N_{3..0}$	if $X(i) > 0$ go to N
JUMP_NEG	1101	i	$N_{7..4}$	$N_{3..0}$	if $X(i) < 0$ go to N

The way the encoding could be optimized is out of the scope of this course. The main idea is to choose an instruction encoding that minimizes the instruction decoder. A very simple example: bit C_{12} that distinguishes between the two possible operations can be directly used to control the arithmetic unit of Fig. 5.14: $f = C_{12}$.

Remember (Comment 5.2) that the minimum number of bits to encode the instructions is 15. The proposed encoding (Table 5.9) needs 16 bits and, as will be seen, the instruction decoder will amount to a few logic gates. Thus, it seems to be a good solution.

5.5.2 Instruction Decoder

The control signals of the five components must be generated in function of the 16 instruction bits $C_{15..0}$.

5.5.2.1 Input Selection Component

From Fig. 5.13a, Table 5.4, and Table 5.9 the following relations are deduced:

$$input\_control = C_{14..13}, j = C_{6..4}, A = C_{11..4}. \tag{5.1}$$

5.5.2.2 Computation Resources

From Fig. 5.14 and Table 5.9 the following relation is deduced:

$$f = C_{12}. \tag{5.2}$$

5.5.2.3 Output Selection Component

From Fig. 5.15, Table 5.5, and Table 5.9 the following equations are generated:

$$out\_en = C_{15} \cdot \overline{C_{14}}\ , out\_sel = C_{13}, i = C_{10..8}, A = C_{7..0}. \tag{5.3}$$

5.5.2.4 Register Bank

From Fig. 5.16, Table 5.6, and Table 5.9 the following equations are deduced:

$$write\_reg\ =\ \overline{C_{15}}\ , i = C_{11..8}, j = C_{7..4}, k = C_{3..0}. \tag{5.4}$$

5.5.2.5 Go To Component

From Fig. 5.17, Table 5.7, and Table 5.9 the following relations are deduced:

$$numb\_sel = C_{15..12}, N = C_{7..0}. \tag{5.5}$$

5.5.3 Complete Circuit

The complete circuit consists of the five components implemented in Sect. 5.4 and of the instruction decoder (5.1–5.5). It is shown in Fig. 5.18.

The following VHDL model describes the circuit of Fig. 5.18. The inputs are the eight 8-bit input ports, the 16-bit instruction, *reset*, and *clock* (not represented in Fig. 5.18). The outputs are the eight 8-bit output ports and *number* (the instruction address):

```
library ...
entity processor is
port (
```

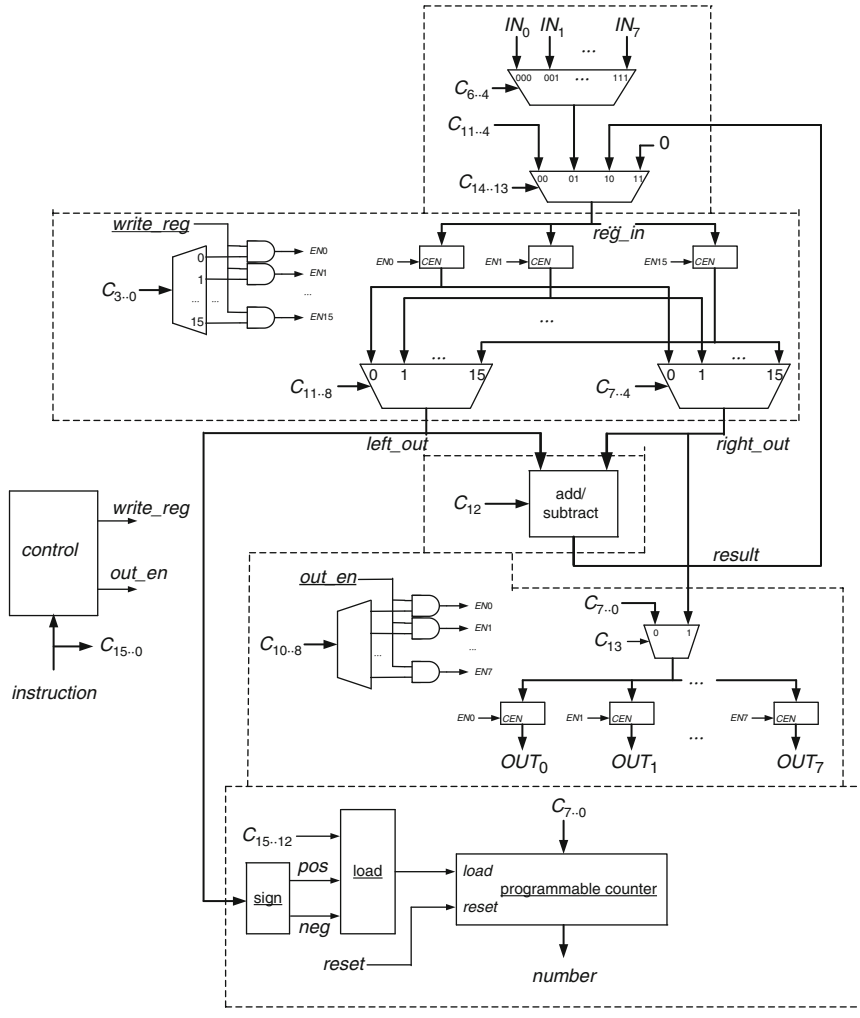


Fig. 5.18 Complete processor

```
    IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7:
       in std_logic_vector(m-1 downto 0);
    instruction: in std_logic_vector(15 downto 0);
    clk, reset: in std_logic;
    OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7:
       out std_logic_vector(m-1 downto 0);
    number: inout std_logic_vector(7 downto 0)
);
end processor;
```

The architecture instantiates and connects the five components, and includes the Boolean equations that define *out_en* and *write_reg*:

```
architecture structure of processor is
 signal write_reg, out_en: std_logic;
```

```
 signal result, reg_in, left_out, right_out:
    std_logic_vector(m-1 downto 0);
component input_selection is
port (
   IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7:
     in std_logic_vector(m-1 downto 0);
   A, result: in std_logic_vector(m-1 downto 0);
   j: in std_logic_vector(2 downto 0);
   input_control: in std_logic_vector(1 downto 0);
   to_reg: out std_logic_vector(m-1 downto 0)
);
end component;
component computation_resources is ...
component output_selection is ...
component register_bank is ...
component go_to is ...
begin
   comp1: input_selection
     port map (IN0 => IN0, IN1 => IN1, IN2 => IN2, IN3 => IN3,
       IN4 => IN4, IN5 => IN5, IN6 => IN6, IN7 => IN7,
       A => instruction(11 downto 4),
       result => result, j => instruction(6 downto 4),
       input_control => instruction(14 downto 13),
       to_reg => reg_in);
   comp2: computation_resources
     port map (left_in => left_out, right_in => right_out,
      f => instruction(12), result => result);
   comp3: output_selection
     port map (A => instruction(7 downto 0), reg => right_out,
     clk => clk, out_en => out_en,
     out_sel => instruction(13),
     i => instruction(10 downto 8),  OUT0 => OUT0,
     OUT1 => OUT1, OUT2 => OUT2, OUT3 => OUT3,
     OUT4 => OUT4, OUT5 => OUT5, OUT6 => OUT6,
     OUT7 => OUT7);
   comp4: register_bank
     port map (reg_in => reg_in, clk => clk,
     write_reg => write_reg,
     i => instruction(11 downto 8),
     j => instruction(7 downto 4),
     k => instruction(3 downto 0), left_out => left_out,
     right_out => right_out);
   comp5: go_to
    port map (N => instruction(7 downto 0), data => left_out,
      clk => clk, reset => reset,
      numb_sel => instruction(15 downto 12),
      number => number);
   --Boolean equations:
   out_en <= instruction(15) AND NOT(instruction(14));
   write_reg <= NOT(instruction(15));
end structure;
```

5.6 Test

To check the working of the processor, test benches that execute programs (temperature controller, chronometer) are defined. For that a VHDL model of the memory that stores the program must be generated.

First consider the temperature control program (Sect. 5.2). A package *program* that includes the definition of a set of constant binary vectors corresponding to the instruction type names (mnemonics) and the program with the encoding of Table 5.9 is defined:

```
library ....
package program is
   -------------------------------------------------
   ----INSTRUCTION SET--------------------
   -------------------------------------------------
   constant ASSIGN_VALUE: std_logic_vector(3 downto 0):= "0000";
   constant DATA_INPUT: std_logic_vector(3 downto 0):= "0010";
   constant DATA_OUTPUT: std_logic_vector(3 downto 0) := "1010";
   constant OUTPUT_VALUE: std_logic_vector(3 downto 0) := "1000";
   constant OPERATION_ADD: std_logic_vector(3 downto 0) := "0100";
   constant OPERATION_SUB: std_logic_vector(3 downto 0) := "0101";
   constant JUMP: std_logic_vector(3 downto 0) := "1110";
   constant JUMP_POS: std_logic_vector(3 downto 0) := "1100";
   constant JUMP_NEG: std_logic_vector(3 downto 0) := "1101";
   -------------------------------------------------
   ----PROGRAM MEMORY DEFINITION AND CONTENTS -------------------
   -------------------------------------------------
   type program_memory is
     array (0 to 255) of std_logic_vector(15 downto 0);
 constant temperature_control: program_memory := (
    ASSIGN_VALUE & x"0A" & x"5",
    DATA_INPUT & "0000" & x"0" & x"0",
    DATA_INPUT & "0000" & x"1" & x"1",
    OPERATION_SUB & x"0" & x"1" & x"4",
    JUMP_NEG & x"4" & x"07",
    JUMP_POS & x"4" & x"09",
    JUMP & "0000" & x"0A",
    OUTPUT_VALUE & x"0" & x"01",
    JUMP & "0000" & x"0A",
    OUTPUT_VALUE & x"0" & x"00",
    DATA_INPUT & "0000" & x"2" & x"3",
    DATA_INPUT & "0000" & x"2" & x"2",
    OPERATION_SUB & x"2" & x"3" & x"4",
    OPERATION_SUB & x"4" & x"5" & x"4",
    JUMP_NEG & x"4" & x"0B",
   JUMP & "0000" & x"01",
   x"0000", x"0000", x"0000", ...);
end program;
```

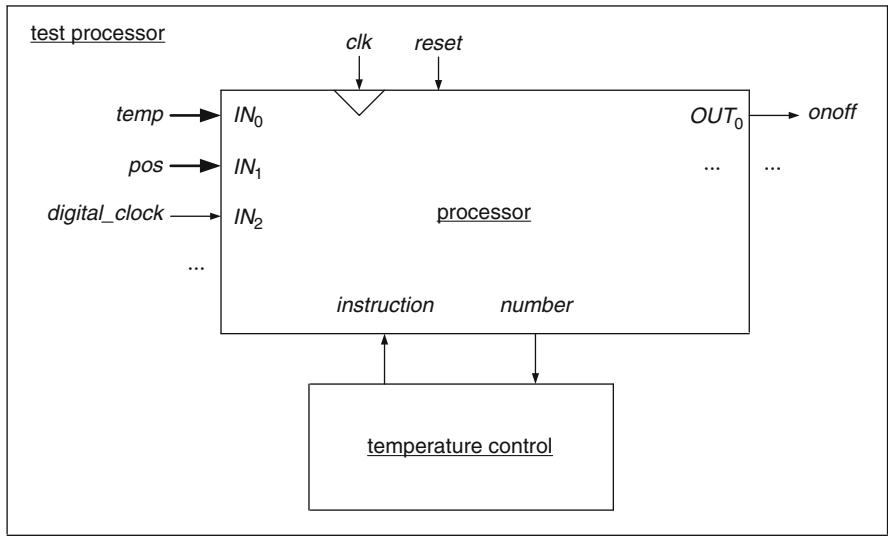


Fig. 5.19 Test bench (temperature control)

Then a test bench is defined (Fig. 5.19). It includes the processor and the program memory modelled by the constant *temperature_control* defined in the package *program*.

The following VHDL model describes the test bench of Fig. 5.19. The architecture contains the instantiation of the processor (*dut*: device under test), the generation of a digital clock that expresses the elapsed time in tenth of seconds, the generation of a *clk* signal with a period of 20 ns, the assignment of external signals *temp*, *pos*, and *digital_clock* to input ports and the assignment of the external signal *onoff* to an output port. A process defines the values of signals *reset*, *pos*, and *temp*. Initially *pos* = 20° (14 in hexadecimal) and *temp* = 16°, so that the output must be equal to *ON*; then *temp* = 20° (15 hex.) and the output must be equal to *OFF*; the next value of *temp* is 20° (14 hex.) so that the output doesn't change; finally *temp* = 19° (13 hex.) and the output must be equal to *ON*:

```
library ...
use work.main_parameters.all;
use work.program.all;
entity test_processor is end test_processor;
architecture test of test_processor is
   component processor is ... end component;
   signal IN0, IN1, IN2, ... : std_logic_vector(m-1 downto 0);
   signal instruction: std_logic_vector(15 downto 0);
   signal clk: std_logic := '0';
   signal reset: std_logic;
   signal OUT0, ... : std_logic_vector(m-1 downto 0);
   signal number: std_logic_vector(7 downto 0);
   signal temp, pos: std_logic_vector(m-1 downto 0);
   signal digital_clock: std_logic_vector(m-1 downto 0) :=
       (others => '0');
   signal onoff: std_logic;
begin
```

```
   dut: processor
     port map(IN0 => IN0, IN1 => IN1, IN2 => IN2, ...,
     instruction => instruction, clk => clk, reset =>  reset,
     OUT0 => OUT0, OUT1 => OUT1, ... , number => number);
     digital_clock <= digital_clock + "00000001" after 100 ns;
     instruction <=
        temperature_control(conv_integer(number));
     clk <= not(clk) after 10 ns;
     IN0 <= temp; IN1 <= pos; IN2 <= digital_clock;
     onoff <= OUT0(0);
--inputs reset, pos, temp
process
begin
     reset <= '1';
     pos <= x"14";
     temp <= x"10";
     wait for 110 ns;
     reset <= '0';
     wait for 2000 ns;
     temp <= x"15";
     wait for 2000 ns;
     temp <= x"14";
     wait for 2000 ns;
     temp <= x"13";
     wait;
 end process;
end test;
```

The simulation result of the test bench is shown in Fig. 5.20.

As a second test bench the chronometer (Sect. 5.1) is implemented. As before a set of constant binary vectors corresponding to the instruction type names (mnemonics) is previously defined. Then the program with the encoding of Table 5.9 is defined by a constant *chronometer*:

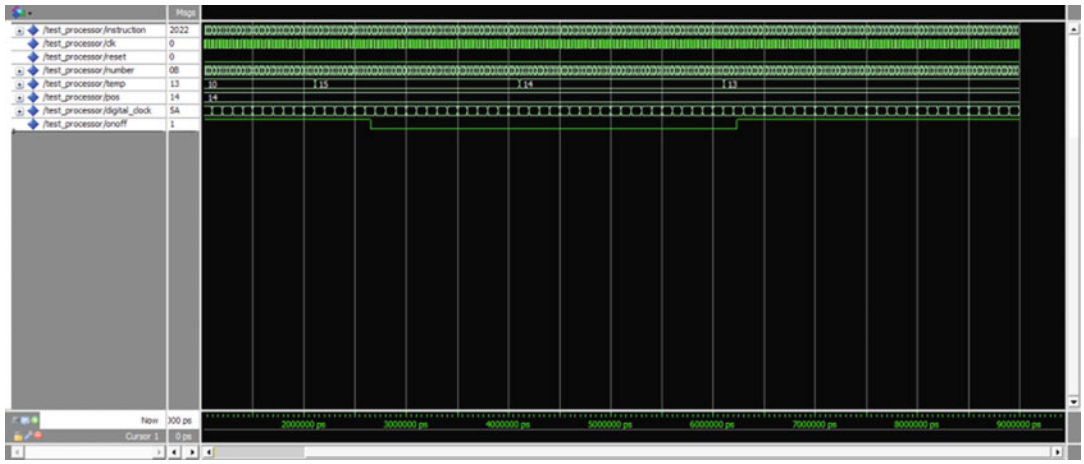


Fig. 5.20 Temperature control simulation (ModelSim Starter Edition, courtesy of Mentor Graphics)

```
constant chronometer: program_memory := (
ASSIGN_VALUE & x"01" & x"3",
DATA_INPUT & "0000" & x"0" & x"0",
JUMP_POS & x"0" & x"06",
DATA_INPUT & "0000" & x"1" & x"0",
JUMP_POS & x"0" & x"09",
JUMP & "0000" & x"01",
ASSIGN_VALUE & x"00" & x"2",
DATA_OUTPUT & x"0" & x"2" & "0000",
JUMP & "0000" & x"01",
DATA_INPUT & "0000" & x"2" & x"0",
JUMP_POS & x"0" & x"01",
DATA_INPUT & "0000" & x"3" & x"1",
JUMP_POS & x"1" & x"0B",
DATA_INPUT & "0000" & x"3" & x"1",
JUMP_POS & x"1" & x"10",
JUMP & "0000" & x"0D",
OPERATION_ADD & x"2" & x"3" & x"2",
DATA_OUTPUT & x"0" & x"2" & "0000",
JUMP & "0000" & x"09",
x"0000", x"0000", x"0000", x"0000", ... );
```

The test bench circuit is shown in Fig. 5.21. It includes the processor and the program memory modelled by the constant *chronometer*. The architecture contains the instantiation of the processor, the generation of a *clk* signal with a period of 20 ns, the generation of an another periodic signal *ref* with a period of 200 ns greater than the *clk* period (Comment 5.1), the assignment of external signals *time_reset*, *start*, *stop*, and *ref* to input ports, and the assignment of the external signal *current_time* to an output port. A process defines the values of signals *reset*, *time_reset*, *start*, and *stop*:

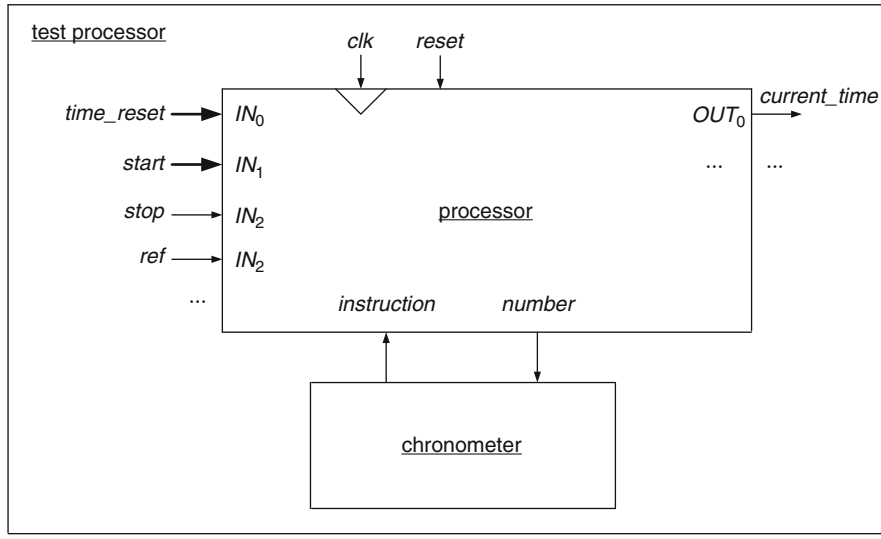


Fig. 5.21 Test bench (chronometer)

```
dut: processor
port map(
  IN0 => IN0, IN1 => IN1, IN2 => IN2, IN3 => IN3, IN4 => IN4,
  IN5 => IN5, IN6 => IN6, IN7 => IN7, instruction => instruction,
  clk => clk, reset => reset, OUT0 => OUT0, OUT1 => OUT1,
  OUT2 => OUT2, OUT3 => OUT3, OUT4 => OUT4, OUT5 => OUT5,
  OUT6 => OUT6, OUT7 => OUT7, number => number);
clk <= not(clk) after 10 ns;
ref <= not(ref) after 100 ns;
IN0 <= time_reset;
IN1 <= start;
IN2 <= stop;
IN3 <= "0000000" & ref;
current_time <= OUT0;
process
begin
   reset <= '1';
   time_reset <= "00000000";
   start <= "00000000";
   stop <= "00000000";
   wait for 100 ns;
   reset <= '0';
   time_reset <= "00000000";
   start <= "00000000";
   stop <= "00000000";
   wait for 100 ns;
   time_reset <= "00000001";
   start <= "00000000";
   stop <= "00000000";
   wait for 200 ns;
   time_reset <= "00000000";
   start <= "00000000";
   stop <= "00000000";
   wait for 200 ns;
   time_reset <= "00000000";
   start <= "00000001";
   stop <= "00000000";
   wait for 400 ns;
   time_reset <= "00000000";
   start <= "00000000";
   stop <= "00000000";
   wait for 4000 ns;
   time_reset <= "00000000";
   start <= "00000000";
   stop <= "00000001";
   wait for 400 ns;
   time_reset <= "00000000";
   start <= "00000000";
   stop <= "00000000";
```

```
    wait for 1400 ns;
    time_reset <= "00000001";
    start <= "00000000";
    stop <= "00000000";
    wait for 200 ns;
    time_reset <= "00000000";
    start <= "00000000";
    stop <= "00000000";
    wait for 200 ns;
    time_reset <= "00000000";
    start <= "00000001";
    stop <= "00000000";
    wait for 400 ns;
    time_reset <= "00000000";
    start <= "00000000";
    stop <= "00000000";
    wait for 4000 ns;
    time_reset <= "00000000";
    start <= "00000000";
    stop <= "00000001";
    wait for 400 ns;
    time_reset <= "00000000";
    start <= "00000000";
    stop <= "00000000";
    wait;
end process;
```

The simulation result of the test bench is shown in Fig. 5.22.

Final Comment

As mentioned above (Preface), this course constitutes a previous step toward other technical disciplines such as computer architecture and embedded systems. In particular, this chapter aimed

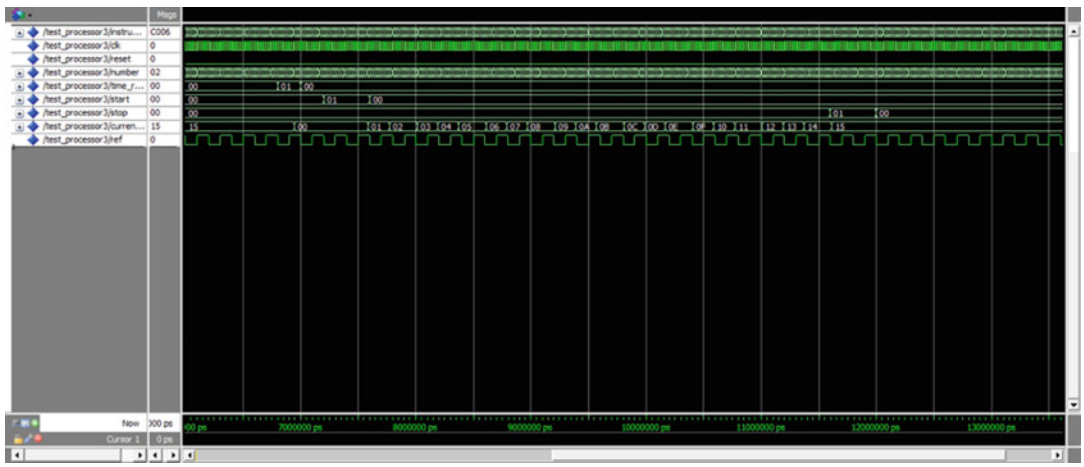


Fig. 5.22 Chronometer simulation (ModelSim Starter Edition, courtesy of Mentor Graphics)

at establishing the relation between digital systems and processors. Many books about the design of processors are available, for example Hamacher et al. (2011) with the description of several commercial processors (Nios, ARM, IA-32, ColdFire) or Harris and Harris (2013).

References

Hamacher C, Vranesic Z, Zaky S, Manjikian N (2011) Computer organization and embedded systems. McGraw-Hill, New York

Harris D, Harris S (2013) Digital design and computer architecture. Morgan Kaufmann, Waltham

Design Methods

6

In Chap. 5 the VHDL model of a complete digital system—a processor—has been generated and the simulation of two test benches has been presented. In this chapter, several comments about the used design method will be made and several alternative options will be proposed.

The circuit descriptions generated in Chap. 5 are register transfer level (RTL, Sect. 7.3) models. Their characteristic is that they include a clock signal and that they define the working of the corresponding circuit cycle by cycle. So, the model explicitly includes the scheduling of the operations. This is the type of model that commercial logic synthesis tools can automatically translate to a physical description based on library components such as gates, flip-flops, and multiplexers.

The same hardware description language (VHDL) has been used to define all RTL descriptions. As already mentioned before, the description of circuits by means of formal languages is a current trend in digital system development, and it is a central aspect of this course. VHDL is a standard, consolidated, widely accepted language, but there are other hardware description languages, for example Verilog and SystemC.

6.1 Structural Description

A summary of the method that has been used to design the processor of Chap. 5 is shown in Fig. 6.1. The starting point was a functional specification. Then, this functional specification has been translated to a block diagram made up of five components: *input selection*, *output selection*, *register bank*, *computational resources*, and *go to*. Each component has been defined by a functional specification. The next step was the definition of an RTL model of each block. Finally, the five components, plus a very simple control unit, have been assembled. In this way a complete RTL description of the processor has been obtained. It remains to synthesize and to implement the complete circuit.

The circuit has been implemented within an xc3s500e FPGA device of Xilinx. The logic synthesis results obtained with ISE Design Suite 14.3 are shown in Table 6.1.

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_6

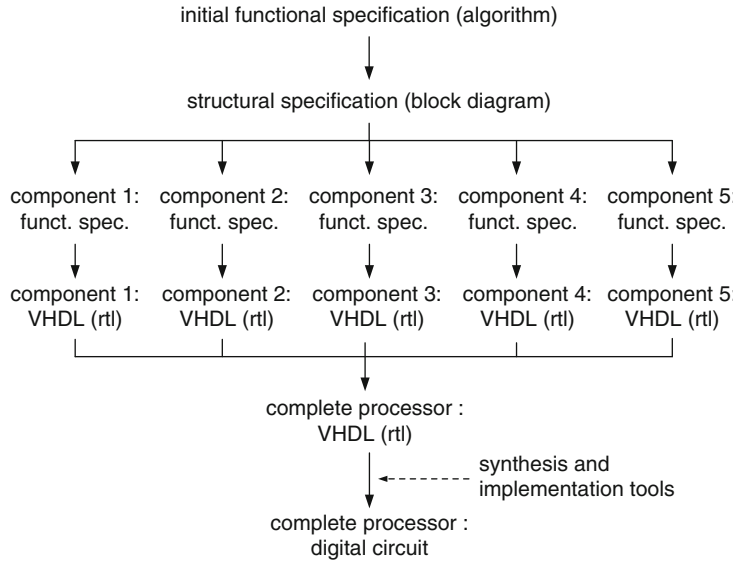


Fig. 6.1 Design method (structural description)

Table 6.1 Synthesis results (structural description)

Component	Number
(8 by 8)-bit ROM	1
8-Bit adder/subtractor	1
8-Bit up counter	1
Flip-flop	192
4-Bit comparator	15
1-Bit MUX16-1	16
8-Bit MUX4-1	1
8-Bit MUX8-1	1

6.2 RTL Behavioral Description

Another design method (Fig. 6.2) could have been used. Starting from the initial functional specification (Algorithm 5.3) an RTL model can be directly generated. Then commercial electronic design automation (EDA) tools are used to synthesize and implement the corresponding circuit.

To translate the functional description of Algorithm 5.3 to a VHDL model, a first step is the generation of a package that defines a set of constants, among others the instruction codes:

```
library ···
package main_parameters2 is
   constant m: natural := 8; -- m-bit processor
   constant zero: std_logic_vector(m-1 downto 0)
     := conv_std_logic_vector(0, m);
   constant one: std_logic_vector(m-1 downto 0)
     := conv_std_logic_vector(1, m);
   constant ASSIGN_VALUE: std_logic_vector(3 downto 0) := "0000";
   constant DATA_INPUT: std_logic_vector(3 downto 0) := "0010";
   constant DATA_OUTPUT: std_logic_vector(3 downto 0) := "1010";
```

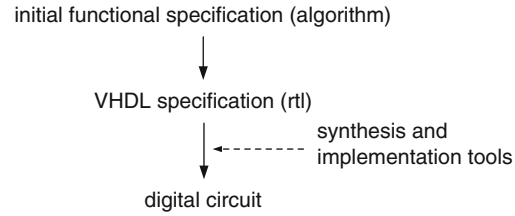


Fig. 6.2 Design method (behavioral description)

```
   constant OUTPUT_VALUE: std_logic_vector(3 downto 0) := "1000";
   constant OPERATION_ADD: std_logic_vector(3 downto 0) := "0100";
   constant OPERATION_SUB: std_logic_vector(3 downto 0) := "0101";
   constant JUMP: std_logic_vector(3 downto 0) := "1110";
   constant JUMP_POS: std_logic_vector(3 downto 0) := "1100";
   constant JUMP_NEG: std_logic_vector(3 downto 0) := "1101";
end main_parameters2;
```

The entity declaration defines the processor inputs and outputs (the same entity declaration as before):

```
library ···
entity processor2 is
port (
   IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7:
     in std_logic_vector(m-1 downto 0);
   instruction: in std_logic_vector(15 downto 0);
   clk, reset: in std_logic;
   OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7:
     out std_logic_vector(m-1 downto 0);
   number: inout std_logic_vector(7 downto 0)
);
end processor2;
```

The architecture includes several type definitions and several signal declarations. The set of memory elements is modelled by an array *X*, the set of input ports is modelled by an array *InputPort*, and the set of output ports is modelled by an array *OutputPort*. The first part of the architecture body contains connections and index definitions. The main part of the architecture is a process directly deduced from the functional specification (Algorithm 5.3).

The main difference with the functional specification is the explicit synchronization of the operations. The process is sensible to the *clk* and *reset* input signals, and the timing specification (or scheduling) is the following: a new instruction execution is started on every positive edge of *clk* and the execution is completed before the next positive edge. Thus, it is an RTL model, not just a functional specification:

```
architecture behavior of processor2 is
   type memory is array (0 to 15) of std_logic_vector(m-1 downto 0);
   signal X: memory;
   type IOPort is array (0 to 7) of std_logic_vector(m-1 downto 0);
   signal InputPort, OutputPort: IOPort;
   signal i, j, k: natural;
   signal onoff: std_logic;
begin
```

```
  InputPort(0) <= IN0; InputPort(1) <= IN1; InputPort(2) <= IN2;
  InputPort(3) <= IN3; InputPort(4) <= IN4; InputPort(5) <= IN5;
  InputPort(6) <= IN6; InputPort(7) <= IN7;
  OUT0 <= OutputPort(0); OUT1 <= OutputPort(1);
  OUT2 <= OutputPort(2); OUT3 <= OutputPort(3);
  OUT4 <= OutputPort(4); OUT5 <= OutputPort(5);
  OUT6 <= OutputPort(6); OUT7 <= OutputPort(7);
  i <= conv_integer(instruction(11 downto 8));
  j <= conv_integer(instruction(7 downto 4));
  k <= conv_integer(instruction(3 downto 0));
  onoff <= OutputPort(0)(0);
  process(clk, reset)
  begin
    if reset = '1' then number <= zero;
    elsif clk'event and clk = '1' then
      case instruction(15 downto 12 ) is
       when ASSIGN_VALUE => X(k) <= instruction(11 downto 4);
         number <= number + one;
       when DATA_INPUT => X(k) <= InputPort(j);
         number <= number + one;
       when DATA_OUTPUT => OutputPort(i) <= X(j);
         number <= number + one;
       when OUTPUT_VALUE => OutputPort(i) <= instruction(7 downto 0);
         number <= number + one;
       when OPERATION_ADD => X(k) <= X(i) + X(j);
         number <= number + one;
       when OPERATION_SUB => X(k) <= X(i) - X(j);
         number <= number + one;
       when JUMP => number <= instruction(7 downto 0);
       when JUMP_POS =>
         if (X(i)(m-1) = '0') and (X(i) /= zero) then
           number <= instruction(7 downto 0);
         else number <= number + one; end if;
       when JUMP_NEG =>
         if X(i)(m-1) = '1' then
           number <= instruction(7 downto 0);
         else number <= number + one; end if;
       when others => null;
       end case;
    end if;
  end process;
end behavior;
```

Comment 6.1

The preceding process defines a type of digital system that generalizes the concept of finite-state machine: it is a finite-state machine with operations. Its internal states are the instructions numbers, and the instruction type (bits 15 down to 12 of the instruction) defines an operation, for example $X(k)$ <=*instruction*(11 *downto* 4) or $X(k)$ <=$X(i)$ + $X(j)$, as well as the new value of *number* (the next internal state).

Table 6.2 Synthesis results (behavioral description)

Component	Number
(16 by 8)-bit dual-port RAM	2
8-Bit adder/subtractor	2
Flip-flop	72
8-Bit MUX8-1	1

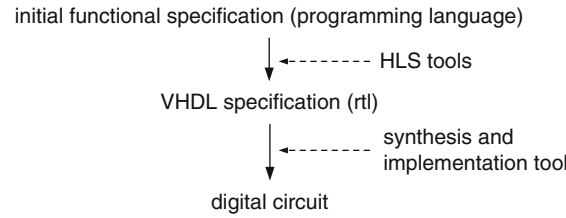


Fig. 6.3 Design method (high-level synthesis)

The logic synthesis results obtained with the same tool (ISE Design Suite 14.3) as before are shown in Table 6.2.

A comparison between the two design methods indicates that the second (Table 6.2) needs much less flip-flops but uses a dual-port RAM. Obviously the difference is the way in which the register bank is synthesized. In the first case it is a low-level implementation with sixteen 8-bit registers (128 flip-flops), an address decoder, and two 8-bit 16-to-1 multiplexers. In the second case, the register bank is a dual-port RAM storing sixteen 8-bit words.

In fact, if in the VHDL model of Chap. 5 the low-level register bank model is replaced by the model of Comment 5.3, the synthesis results would be similar to those of Table 6.2.

From the comparison between the two design methods several conclusions can be drawn; among others:

- Modern EDA tools have the capacity to divide a system into components, starting from a behavioral specification; for example, if the behavioral specification is a finite-state machine with operations (Comment 6.1), the logic synthesis tools will instantiate and interconnect the components that execute the operations, and will extract from the behavioral description the corresponding control unit.
- There is a direct relation between the VHDL (or any other HDL) description and the implementation results; thus, the designer work is not only the generation of a correct (structural or behavioral) description; it is the generation of a description that the chosen EDA tools will translate to an efficient implementation.

6.3 High-Level Synthesis Tools

A more advanced method (Fingeroff 2010; Xilinx 2013) is the use of high-level synthesis (HLS) tools (Fig. 6.3). They permit to translate an initial functional specification in some programming language, for example C, to an RTL specification in some hardware description language. So, the designer is only responsible for this initial specification, and the rest of the design work is performed by automatic design tools.

As an example consider the following functional specification of the processor of Chap. 5. It is a C program that can be translated to an RTL description (VHDL, Verilog, SystemC) with Vivado HLS (the HLS tool of Xilinx). It uses some predefined unsigned integer types (4 bits, 8 bits, and 16 bits).

The functional specification is very similar to Algorithm 5.3, except for the syntax differences between C and VHDL. The code values 0, 2, 10, and so on correspond to the instruction types *ASSIGN_VALUE*, *DATA_INPUT*, *DATA_OUTPUT*, etc., as defined by bits $C_{15..12}$ in Table 5.9. The main conceptual difference between this C description and the VHDL description of Sect. 6.2 is that the former doesn't include a synchronization signal, and the scheduling of the operations will be defined by the HLS tool. The designer can give orders to the HLS tool; for example the pragma *HLS INTERFACE* permits to define the type of input and output interfaces. Other available orders permit to control the way the HLS tool schedules the operations:

```
typedef ap_int<8>  data;
typedef ap_uint<4>  instruction_field;
typedef ap_uint<8>  address;
typedef ap_uint<16>  instruction;

#include ···
void hls_processor (bool reset, data IN0, data IN1, data IN2,
   data IN3, data IN4, data IN5, data IN6, data IN7,
   instruction_field code, instruction_field i, instruction_field j,
   instruction_field k, data * OUT0, data * OUT1, data * OUT2,
   data * OUT3, data * OUT4, data * OUT5, data * OUT6, data * OUT7,
   address * number)
{
   #pragma HLS INTERFACE ap_ctrl_none port=return
   static data X[16];
   static address pc;
   data input_port[8];
   input_port[0] = IN0; input_port[1] = IN1; input_port[2] = IN2;
   input_port[3] = IN3; input_port[4] = IN4; input_port[5] = IN5;
   input_port[6] = IN6; input_port[7] = IN7;
   static data output_port [8];
   ////////////////////////////////////////////////////////////
   ////////////////////FUNCTIONAL SPECIFICATION////////////////////
   ////////////////////////////////////////////////////////////
   if (reset == 1) pc = 0;
   else {
        switch (code)
          {
          case 0: {X[k] = j + 16*i; pc = pc + 1; } break;
          case 2: {X[k] = input_port[k]; pc = pc + 1; }break;
          case 10: {output_port[i] = X[j]; pc = pc + 1; }break;
          case 8: {output_port[i] = k + 16*j; pc = pc + 1; }break;
          case 4: {X[k] = X[i] + X[j]; pc = pc + 1; } break;
          case 5: {X[k] = X[i] - X[j]; pc = pc + 1; } break;
          case 14: {pc = k + 16*j; } break;
          case 12: {if (X[i] > 0) pc = k + 16*j;
                    else pc = pc + 1; } break;
          case 13: {if (X[i] < 0) pc = k + 16*j;
                    else pc = pc + 1; } break;
          }
        }
```

```
    ////////////////////////////////////////////////////////////
    *OUT0 = output_port[0]; *OUT1 = output_port[1];
    *OUT2 = output_port[2]; *OUT3 = output_port[3];
    *OUT4 = output_port[4]; *OUT5 = output_port[5];
    *OUT6 = output_port[6]; *OUT7 = output_port[7];
    *number = pc;
}
```

Many books on C and C++ are available, for example Alfonseca and Sierra (2005) in Spanish or Deitel and Deitel (2015) in English.

References

Alfonseca M, Sierra A (2005) Programación en C/C++. Anaya Multimedia, Madrid
Fingeroff M (2010) High-level synthesis blue book. Xlibris Corporation, Bloomington
Deitel P, Deitel H (2015) C how to programa. Prentice Hall, Upper Saddle River
Xilinx (2013) http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

Physical Implementation 7

This last chapter presents some basic information about manufacturing technologies, as well as about implementation strategies, and synthesis and implementation tools.

7.1 Manufacturing Technologies

All along the course digital circuits have been designed with components such as logic gates, multiplexers, and flip-flops. The result of the design work is a logic diagram of the type shown in Fig. 7.1.

How can such a logic diagram be physically implemented? A first option is to use standard (off the shelf) small-scale, medium-scale, and large-scale integrated circuits, capacitors, resistors, and other passive components, and to interconnect them on a printed circuit board (PCB). An example of digital system implemented in this way is shown in Fig. 7.2.

Another option is to design and manufacture a new specific integrated circuit.

Consider the first option. Two examples of standard small-scale integrated circuits (chips) are shown in Fig. 7.3. The first (Fig. 7.3a) is a chip that integrates four AND2 gates and the second (Fig. 7.3b) is a chip that integrates three NOR3 gates. Figure 7.3 shows the internal logic circuit, a photograph of the package, and the assignment of package pins to internal logic signals. For every chip, the vendor gives to the designer a data sheet that defines the logic and electrical characteristics of the component, the mechanical characteristics of the package, the pin assignment, and so on.

It remains to interconnect the components (standard chips, discrete electrical components, mechanical switches). During the development phase of a new digital system, a prototyping board like that of Fig. 7.4a can be used: the components are plugged into the board holes and the connections are made with wires (Fig. 7.4b). Obviously circuits implemented with this type of connections (plugged-in wires) are not very reliable but are quite similar to the final system and permit to test their interaction with other systems.

Once the prototype has been satisfactorily tested, a PCB is designed and manufactured. It includes the holes within which the chip pins will be inserted (through-hole technology) and the metal tracks that implement the connections. Nowadays surface-mount devices (SMD) are also used; those components can be placed directly onto the surface of the PCB (surface-mount technology). A system similar to that of Fig. 7.2 is obtained. Current EDA tools permit to design the PCB and to simulate the complete system before the manufacturing of the PCB.

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_7

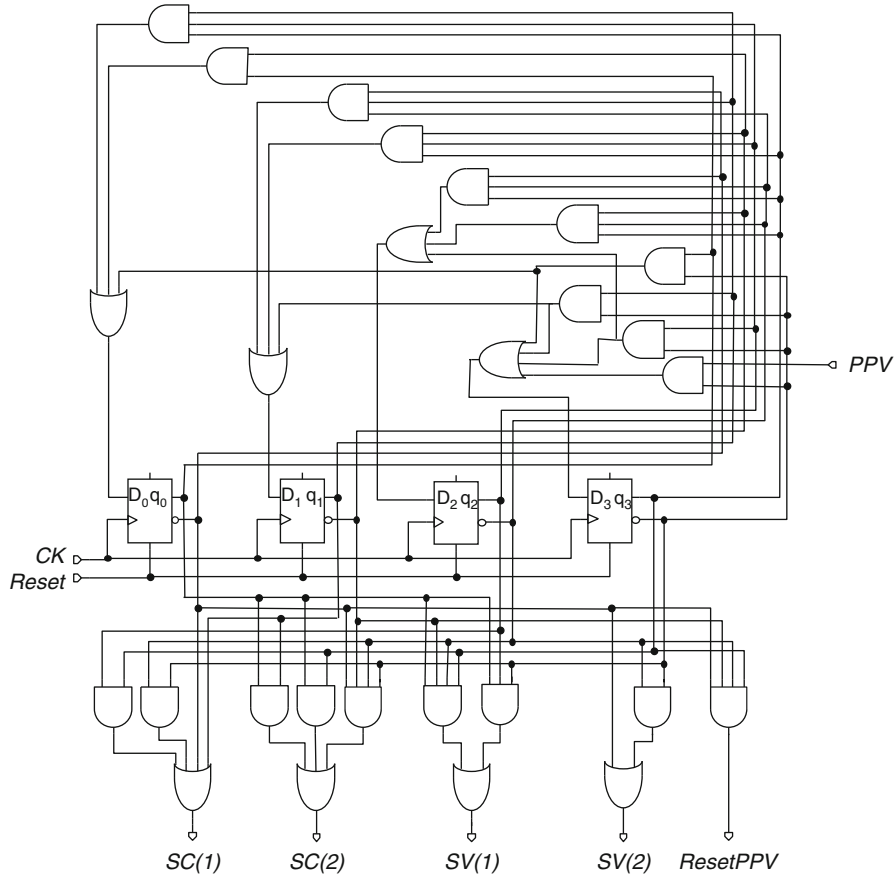


Fig. 7.1 Example of logic diagram

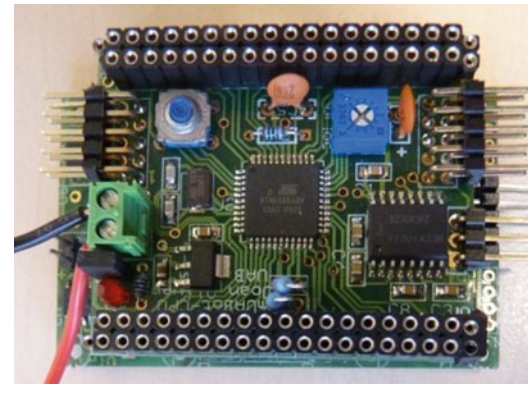


Fig. 7.2 Example of physical implementation

The use of standard components is a convenient option in the case of small circuits. In the case of large circuits, a better option could be the development of a new, large-scale, integrated circuit, a so-called application-specific integrated circuit (ASIC) that integrates most functions of the system under development.

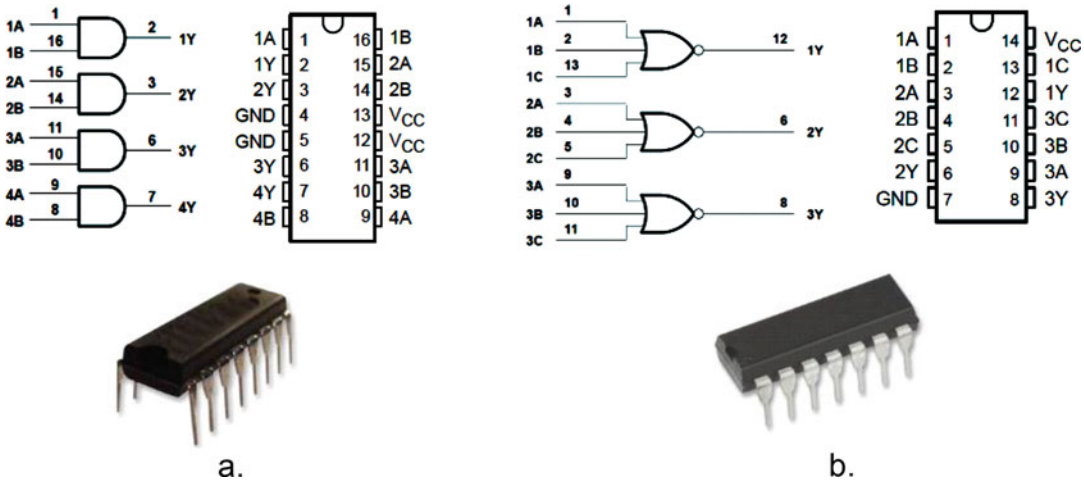


Fig. 7.3 Two examples of small-scale integrated circuits

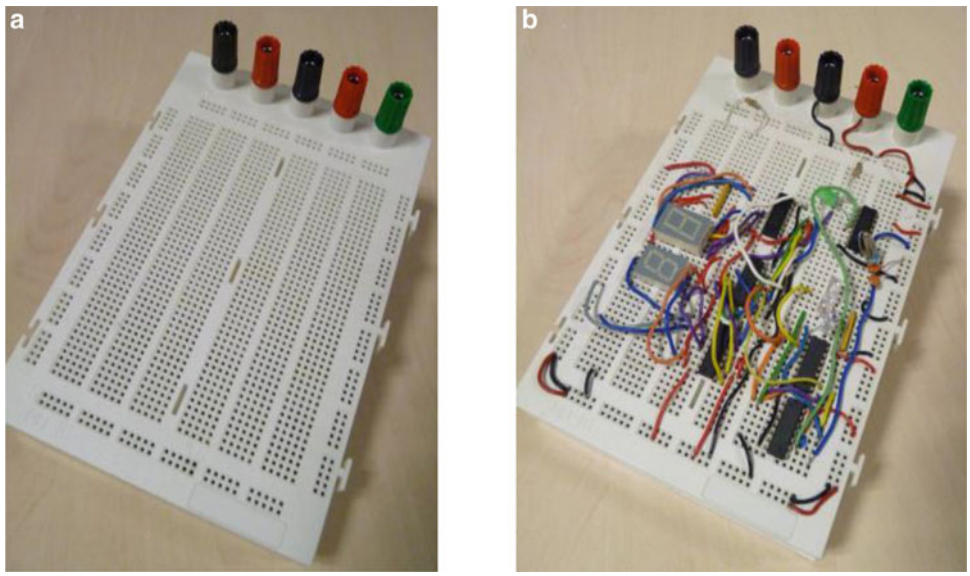


Fig. 7.4 Prototyping board

An integrated circuit (IC) package is shown in Fig. 7.5a. Within the package, there is a small silicon die that integrates the whole circuit. It is connected by very thin wires to small metal tracks. Those tracks are connected to the external pins of the package.

The technology used to manufacture integrated circuits is microelectronics. It permits to integrate electronic circuits on a semiconductor substrate, most often silicon. The fabrication consists of several processes such as oxidation of the silicon slice, deposition of some layer, etching of the deposited layer, iron implementations, and others. Most of those processes need a mask because at each step certain areas must be masked out. As an example, *p*-type transistors are integrated within *n*-type wells that are created by implanting negative ions within the corresponding area. So, a masking technique is used to implant negative ions only within the *p*-type transistor areas.

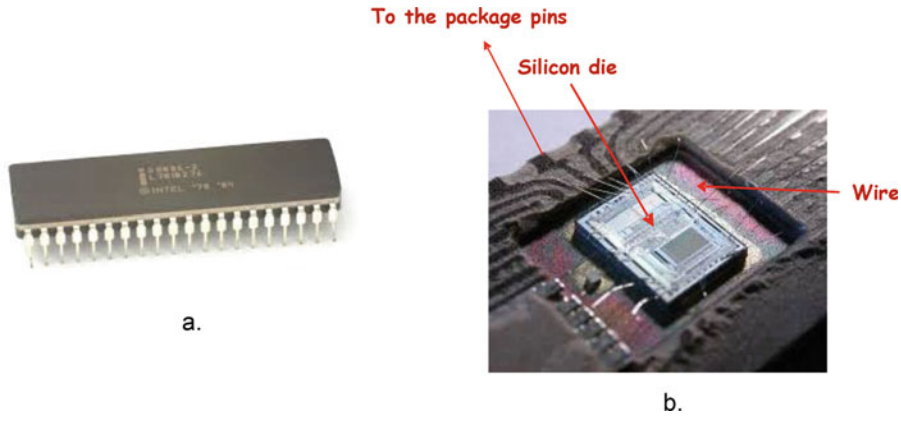


Fig. 7.5 Integrated circuit

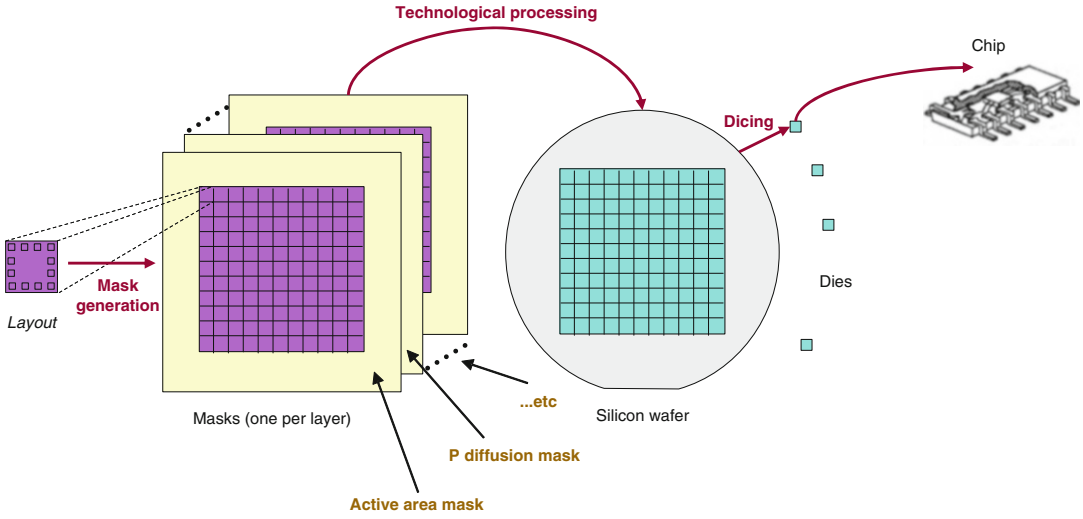


Fig. 7.6 IC manufacturing

The integration density in terms of transistors per square millimeter is very high. As an example, the Intel Core i7 includes more than 700 million transistors within less than 300 mm^2.

The information that an IC production line (a silicon foundry) needs to manufacture a circuit is the layout (Fig. 7.6): it is the geometric information necessary to fabricate the masks. As mentioned above, most fabrication steps need a specific mask. For example, a first mask defines the areas where there are transistors (active areas); another mask defines the areas where there are *p*-type transistors, and so on. Then, using the layout information, a set of masks is fabricated (Fig. 7.6). They are used to process silicon slices (silicon wafers) with a diameter of up to 300 mm. In Fig. 7.6 each square on the wafer surface is a circuit, so that a lot of ICs are manufactured at the same time. Once the wafer has been processed, it must be cut—it's the dicing process—and each die must be encapsulated in a package.

The generation of the layout, from a logic diagram, is not an easy task. As an example, Fig. 7.7b shows the layout of a CMOS inverter whose electrical circuit is shown in Fig. 7.7a, and a cross section

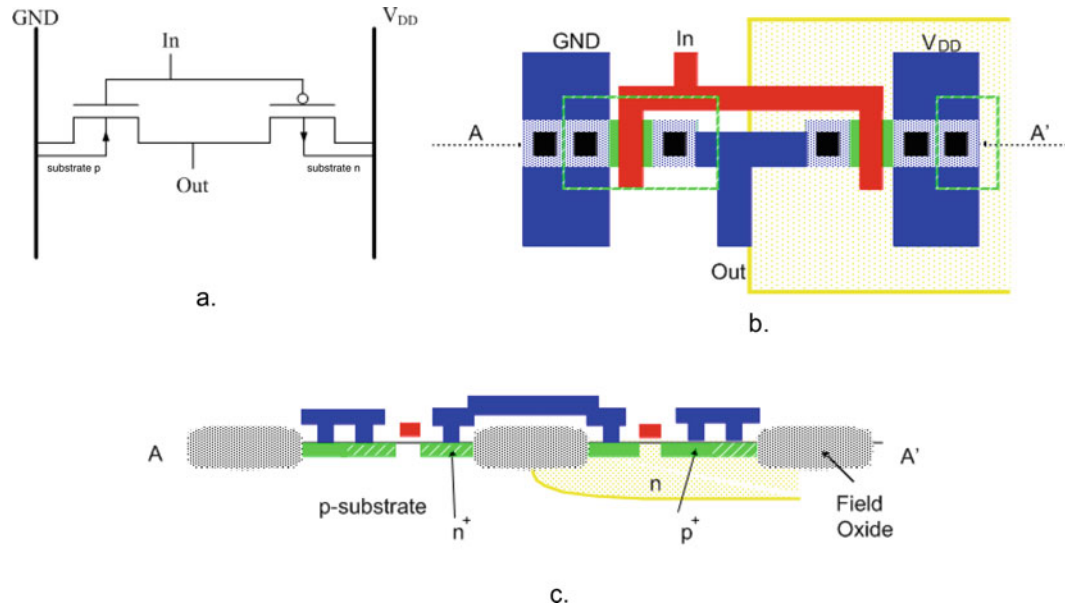


Fig. 7.7 CMOS inverter layout

AA′ of the device to be manufactured is shown in Fig. 7.7c. The circuit consists of only two transistors, an *n*-type transistor and a *p*-type transistor. Each color defines a different mask. For example, the blue areas correspond to a mask used to define the metal tracks. The red areas correspond to polysilicon layers that are the transistor gates; observe that the *n*-type and *p*-type transistor gates are defined by a common polysilicon (red) area that is also the gate input. The green areas correspond to *n*-type or *p*-type diffusion layers that are the sources and drains of the transistors. The yellow areas correspond to *n*-type wells within which *p*-type transistors will be integrated, and so on. A detailed description of the manufacturing process can be found in many books on microelectronics, IC design, manufacturing, etc., for example Chapter 2 of Rabaey et al. (2003).

The conclusion is that the design of a circuit integrating hundreds of thousands of logic gates implies the generation of millions of geometric forms (rectangles, squares), with several colors representing the layer types (metal, polysilicon, diffusion, wells, contact holes between layers, etc.). Furthermore

- There is a set of design rules that must be taken into account, for example the minimum width of a polysilicon layer, the minimum distance between two metal tracks, and many others.
- The size of (at least part of) the geometric forms must be computed; as an example, the delay of a gate directly depends on the size of the corresponding transistors.

How to manage such a complex task? The answer is

- The use of implementation strategies such as standard cell approach, gate array approach, and field programmable gate arrays (FPGA)
- The development and use of electronic design automation (EDA) tools

7.2 Implementation Strategies

Three implementation strategies are described: standard cell approach, mask programmable gate arrays, and FPGA.

7.2.1 Standard Cell Approach

This design method is based on the use of predefined libraries of cells. Those cells are logic components such as gates, flip flops, multiplexers, and registers. The layout of each cell is generated in advance and is stored in a cell library. To make the assembling of the circuit easy, all cells have the same height.

As an example, the layout of a NAND3 gate is shown in Fig. 7.8. It integrates three *n*-type serially connected transistors in the lower part, and three *p*-type transistors connected in parallel in the upper part.

The layout of the circuit is generated by a place and route EDA tool. The cells are placed in rows whose height is the common cell height (Fig. 7.9). Macrocells such as memories, multipliers, and

Fig. 7.8 NAND3 cell

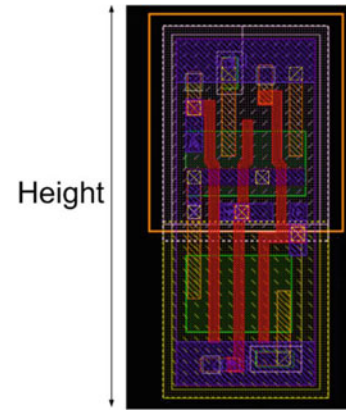
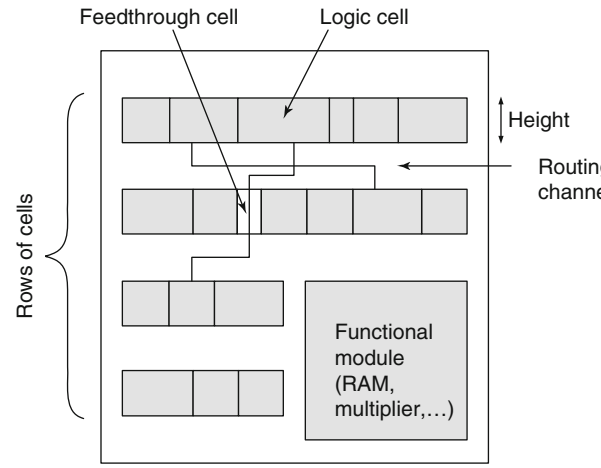


Fig. 7.9 Floor plan



processor cores can also be added. An example of resulting floor plan is shown in Fig. 7.9. It consists of a set of rows of standard cells (previously designed and characterized cells). The space between rows is used to lay out connections. Sometimes feed-through cells (connections) are inserted. In many cases there are also macrocells.

As a matter of fact, in modern standard cell technologies there are no routing channels. The upper metal layers are used to lay out connections on top of the cells.

7.2.2 Mask Programmable Gate Arrays

The gate array approach consists in manufacturing in advance arrays of cells, without regard to the final application, but in such a way that only connections must be added to meet the circuit specification.

The floor plan is similar to that of a standard cell circuit. An example is shown in Fig. 7.10. The floor plan consists of rows of uncommitted cells separated by routing channels. An uncommitted cell is an uncomplete logic component: to get an actual logic component, internal connections must be added. The floor plan of a particular gate array is predefined (cannot be modified). The number of rows, the size of the rows, and the distance between rows are predefined, and so is also the maximum number of cells and the maximum capacity in terms of logic gates. On the other hand, the gate array vendors offer families of components with different gate capacities and different maximum numbers of input and output pins.

An example of uncommitted cell is shown in Fig. 7.11a (layout) and 7.11b (electrical circuit). It contains four *p*-type transistors and four *n*-type transistors. The red tracks are the common gates of the four transistor pairs.

Then (Fig. 7.12) by adding metal connections, a 4-input NOR gate is obtained. The four *p*-type transistors are connected in series and the four *n*-type transistors are connected in parallel between Out and GND.

As in the case of standard cells, in modern gate arrays there are no more routing channels. The connections are laid out on top of the cells. Those channelless gate arrays are sometimes called seas of gates.

7.2.3 Field Programmable Gate Arrays

An FPGA is similar to a gate array but it is user programmable. An example of floor plan is shown in Fig. 7.13. The device consists of a set of programmable basic cells and of programmable connections.

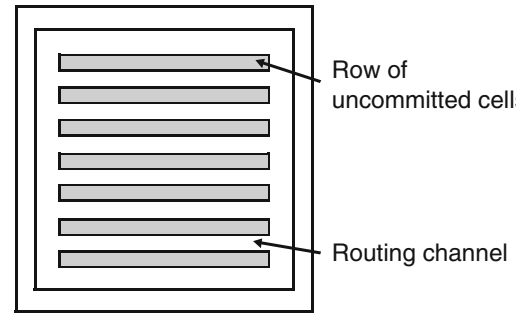


Fig. 7.10 Gate array floor plan

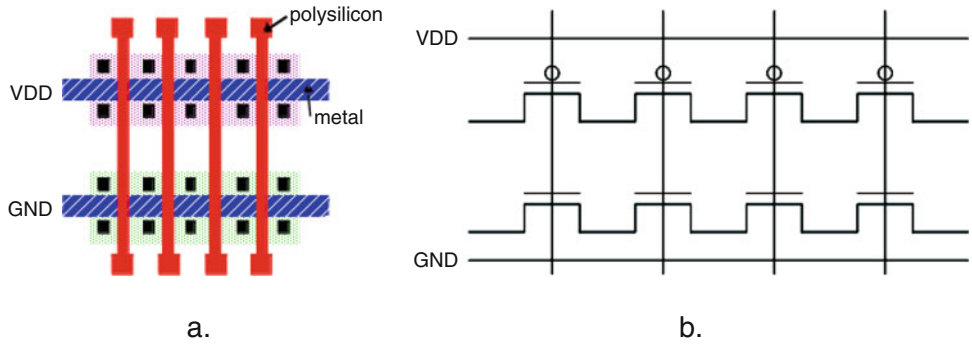


Fig. 7.11 Uncommitted cell

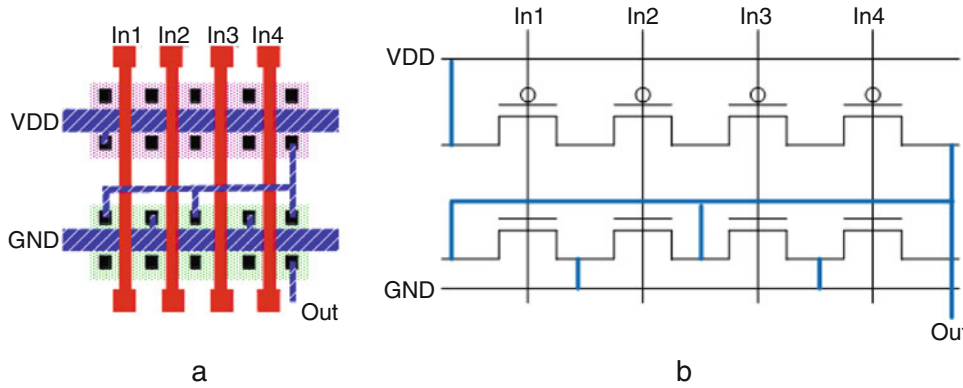
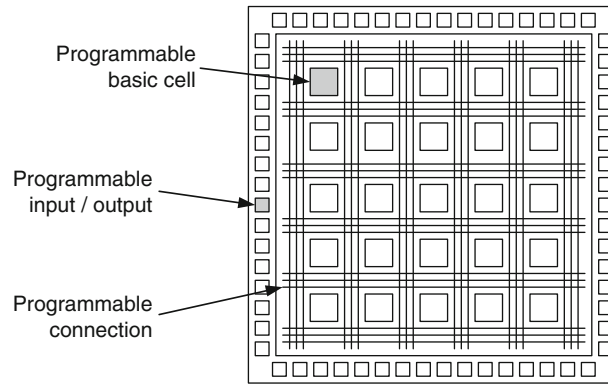


Fig. 7.12 NOR4 gate

Fig. 7.13 FPGA floor plan



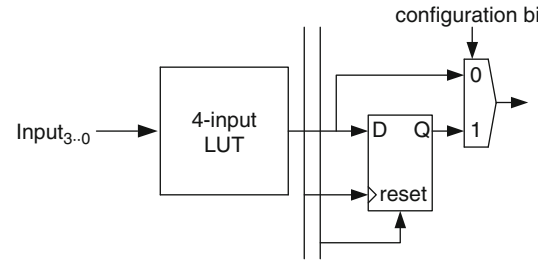


Fig. 7.14 Programmable basic cell

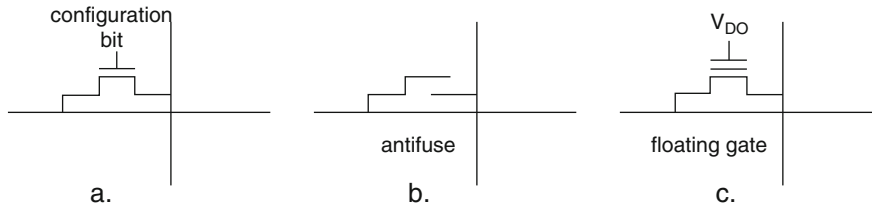


Fig. 7.15 Programmable connections

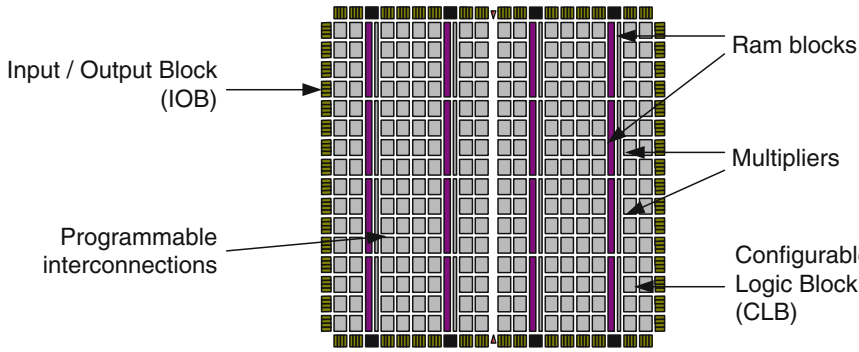


Fig. 7.16 Commercial FPGA (courtesy of Xilinx)

An example of simplified programmable basic cell is shown in Fig. 7.14. It includes a look-up table (LUT), able to implement any 4-variable switching function, and a flip-flop. In function of the value of the configuration bit that controls a MUX2-1, the cell output is equal to the look-up table output or to the flip-flop output.

The connections between cells are also programmable. Three examples of connections between a vertical line and a horizontal line are shown in Fig. 7.15. The first example (Fig. 7.15a) is a transistor controlled by a configuration bit. The second example (Fig. 7.15b) is an anti-fuse: it is a switch, initially open, that can be short-circuited by applying a (relatively) high voltage between its terminals. The third example (Fig. 7.15c) is a floating-gate transistor: according to the amount of electric charges stored on the floating gate, this transistor conducts or doesn't.

So, apart from the basic cell and from the connections, an FPGA also includes a lot of memory elements that store the configuration data, like an underlying static RAM, EEPROM, or OTP ROM (Sect. 4.6.3).

FPGA devices also include macrocells. An example of medium-size commercial FPGA is shown in Fig. 7.16. It includes logic cells (CLB), interface cells (IOB), RAM, and multipliers.

Modern FPGA are very large circuits. They permit to implement complete systems on a chip, including microprocessors and memory blocks. They constitute a very good option to fabricate prototypes. They can also be efficiently used in the case of low-quantity production.

7.3 Synthesis and Physical Implementation Tools

In this section some comments about synthesis and implementation tools are given.

A first point is the definition of the concept of register transfer level (RTL) description.

Definition 7.1 An RTL description is a functional description, for example in some hardware description language, with the following property: they include a clock signal and the operations executed during each clock period are defined.

Several examples of RTL descriptions have been seen before (the programmable timer of Sect. 4.9.1, the processor of Chap. 5). Consider the timer of Sect. 4.9.1. Its description consists of two processes *next_state_registered* and *output_state*. It is a functional description: none of the two processes include references to components such as gates and flip-flops. On the other hand the *next_state_registered* process is synchronized by a clock signal *clk* and defines, on every positive edge of *clk*, the next internal state in function of the input signals *start*, *zero*, and *reference*, and the *output_state* process defines a combinational circuit that defines the 2-bit output signal *operation* in function of the current state.

Once an RTL description has been defined and tested, for example, by simulation, a logic synthesis tool may be used. The synthesis is done in two steps.

- A first step is the generation, from the RTL description, of a technology-independent circuit structure. The circuit is made up of generic gates, flip-flops, etc., without specifying the target technology.
- The second step (technology mapping) is the generation of a circuit made up of cells that belong to a particular library (standard cell, gate array, FPGA). The designer must choose this particular library among all available cell libraries of all IC vendors in function of criteria such as cost, speed, power consumption, or time to market.

After the logic synthesis, the next operation is the physical implementation. The physical implementation tools generate the data necessary to fabricate or to program the circuit using the previously obtained structural description with library components. In the case of an ASIC (standard cell or gate array) the information to be generated is the layout. In the case of an FPGA, the information to be generated is the set of all configuration bits and LUT contents.

In all cases the physical implementation operation consists of at least two steps. One of them is the placement: a place within the circuit floor plan is assigned to every cell of the structural description (in the case of a standard cell approach, a floor plan must have been previously defined). The other operation is the routing: the connections between the placed cells must be laid out or configured.

Reference

Rabaey JM, Chandrakasan A, Nikolic B (2003) Digital integrated circuits: a design perspective. Prentice Hall, Upper Saddle River

Appendix A: A VHDL Overview

This appendix collects the very basics of VHDL language (**V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage) with the goal of providing enough knowledge to read and understand its usage throughout this book and start developing basic hardware models. This appendix intends to teach by examples; consequently, most of them will be related to the modules and topics already addressed in the different chapters of this book.

There are plenty of books and webs available for further VHDL learning, for example Ashenden (1996), Terés et al. (1998), Ashenden and Lewis (2008), and Kafig (2011).

A.1 Introduction to Hardware Description Languages

The current hardware description languages (HDLs) were developed in the 1980s in order to manage the growing complexity of very-large-scale integrated circuits (VLSI). Their main objectives are:

- Formal hardware description and modelling
- Verification of hardware descriptions by means of their computer simulation
- Implementation of hardware descriptions by means of their synthesis into physical devices or systems

Current HDLs share some important characteristics:

- Technology independent: They can describe hardware functionality, but not its detailed implementation on a specific technology.
- Human and computer readable: They are easy to understand and share.
- Ability to model the inherent concurrent hardware behavior.
- Shared concepts and features with high-level structured software languages like C, C++, or ADA.

Hardware languages are expected to model functionalities that once synthesized will produce physical processing devices, while software languages describe functions that once compiled will produce code for specific processors.

Languages which attempt to describe or model hardware systems have been developed at academic level for a long time, but none of them succeeded until the second half of the 1980s.

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9

VHDL emerged from the initiative VHSIC (Very High Speed Integrated Circuit) of the US Department of Defense (DoD) in 1981 and was later joined by Itermetics, IBM, and Texas Instruments. The first version of VHDL was released in 1985 and its success convinced the Institute of Electrical and Electronics Engineers (IEEE) to formalize the language into the standard IEEE 1076-1987, released in 1987 and successively updated by committees in 1993, 2000, 2002, and 2008 (IEEE 1076-2008 or VHDL-2008).

Verilog also made its appearance during the 1980s. It initially focused on ASIC digital-timing simulation, but it evolved to HDL once the VHDL started to gain adepts, projects, and success. Again, it was adopted into a new standard—IEEE 1364-1995—in 1995 by the IEEE.

Verilog and VHDL are the most popular languages covering the same circuit design abstraction levels from behavioral down to RTL and gate. Both languages shared evolutionary paths and right now they are fully interoperable and easy to combine for circuit modeling, simulation, and synthesis purposes.

Other languages like SystemVerilog and SytemC are addressing higher abstraction levels to specify and model more complex hardware/software systems.

This appendix will be exclusively devoted to an overview of VHDL, its basics concepts, sentences, and usage.

A.2 VHDL Main Characteristics

VHDL is derived from its equivalent in software development, the ADA language, which was also created by an initiative of the US-DoD a few years before (1977–1983). As such, both share concepts, syntax, and structures, including the ability for concurrent modelling.

VHDL is a high-level structured language, strongly and statically typed, non-case sensitive, and able to describe both concurrent and sequential behaviors.

A.2.1 Syntax

Like any other computer language, VHDL is based on its own sentences, where reserved words, identifiers, symbols, and literals are combined to write the design units which perform the related hardware models or descriptions.

Reserved words are the specific words used by the language to fix its own syntax indifferent sentences and structures. In the VHDL examples along this appendix, they will be highlighted in bold text.

Identifiers are specific names associated with each language object, structure, data type, or design unit in order to refer to any one of them. A few basic rules to create such identifiers are:

- Allowed character set is {“a”…“z”, “A”…“Z”, “0”…“9”, “_”}.
- First character must be alphabetic.
- Two consecutive “_ _” anywhere or ending with “_” is forbidden.
- VHDL is case insensitive, and identifiers can be of any length.
- Reserved words are forbidden as identifiers.
- Good examples: COUNT, En_10, aBc, X, f123, VHDL, VH_DL, Q0.
- Bad examples: _Ctrl, 2counter, En__1, Begin, q0_, is, type, signal.

Symbols are sets of one or two characters with a specific meaning within the language:

- Operators: + - * / ** ( ) < > = /= >= & …
- Punctuation: . , : ‘’ “” # ;

- Part of expressions or sentences: => := /= >= <= |
- Comments: -- after this symbol, the remaining text until the end of the line will be considered a comment without any modelling effect but just for documentation purposes.

Literals are explicit data values of any valid data type which can be written in different ways:

- Base#Value#: 2#110_1010#, 16#CA#, 16#f.ff#e+2.
- Individual characters: "a", "A", "@", "?"
- Strings of characters: "Signal value is", "11010110", "#Bits:".
- Individual bits: "0", "1".
- Strings of bits (Base"Value"): X"F9", B"1111_1001" (both represent the same value).
- Integers and reals in decimal: 12, 0, 2.5, 0.123E-3.
- Time values: 1500 fs, 200 ps, 90 ns, 50 µs, 10 ms, 5 s, 1 min, 2 h.
- Any value from a predefined enumerated data type or physical data type.

A.2.2 Objects

Objects are language elements with specific identifiers and able to contain any value from their associated data type. Any object must be declared to belong to a specific data type, the values of which are exclusively allowed for such object. There are four main objects types in VHDL: constants, variables, signals, and files.[1] All of them must be declared beforehand with the following sentence:

```
<Object type> <identifier>: <data type> [:=Initial value];
```

The initial value of any object is, by default, the lowest literal value of its related data type, but the user can modify it using the optional clause ":= initial value".

Let's show a few object declarations:

```
constant PI : real := 3.1415927;
constant WordBits : natural := 8;
constant NumWords : natural := 256;
constant NumBits : natural := WordBits * NumWords;
variable Counter : integer :=0;
variable Increment : integer;
signal Clk : bit := '1';
```

Constants and ***variables*** have the same meaning as in software languages. They are pieces of memory, with a specific identifier, which are able to contain literal values from the specified data type. In the case of constants, once a value is assigned it remains fixed all the time (in the above examples constants are assigned in the declaration). A variable changes its value immediately after each new assignment. Next sentences are examples of variable assignments:

```
Increment :=2;
Counter := Counter + Increment;
```

[1] File object will not be considered along this introduction to VHDL.

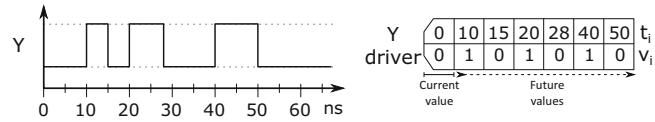


Fig. A.1 Waveform assigned to signal Y and its related Driver

Signals are the most important objects in VHDL, and they are used to describe connections between different components and processes in order to establish the data flow between resources that model the related hardware.

Signals must be able to reflect their value evolution across time by continuously collecting current and further values as shown in the next code box examples. For a "Signal" object, in comparison with "Constants and Variables", a more complex data structure is required, as we need to store ***events***, which are pairs of "***value*** (v_i) − ***time*** (t_i)" in chronological order (the related signal reaches the value v_i at time t_i). Such a data structure is called a ***Driver***, as shown in Fig. A.1 which reflects the Y signal waveform assignment below:

```
Reset <= '1'; -- Assigns value '1' to signal Reset.
-- Next sentence changes Clock signal 10ns later.
Clock <= not Clock after 10ns;
-- Next sentence projects on signal Y a waveform of
-- different values at different times.
Y <= '0', '1' after 10 ns, '0' after 15 ns, '1' after 20 ns, '0' after 28 ns, '1' after
40 ns, '0' after 50 ns;
```

We will come back to signal driver management for event-driven VHDL simulation flow and cycle.

Comment A.1
Observe that while constant and variable assignments use the symbol ":=", the symbol used for signal assignments is "<=".

A.2.3 Data Types

VHDL is a strongly typed language: any object must belong to a specific previously defined data type and can only be combined with expressions or objects of such data type. For this reason, conversion functions among data types are usual in VHDL.

A data type defines a set of fixed and static values: the literals of such data type. Those are therefore the only ones that the objects of such a data type can contain.

There are some basic data types already predefined in VHDL, but new user-defined data types and subtypes are allowed. This provides a powerful capability for reaching higher abstraction levels in VHDL descriptions. In this appendix, just a subset of predefined (integer, Boolean, bit, character, bit_vector, string, real, time) and a few user-defined data types will be used. Most of the data types are defined in "Packages" to allow their easy usage in any VHDL code. VHDL intrinsic data types are declared in the "Standard Package" shown in the next code box with different data types: enumerated (Boolean, bit, character, severity_level), ranged (integer, real, time), un-ranged arrays (string, bit_vector), or subtypes (natural, positive):

```
Package standard is
  type boolean is (false, true);
  type bit is ('0', '1');
  type character is (NUL, SOH, STX,..., ", '!', '#', '$'
       ,..., '0', '1', '2', '3', '4', '5', '6', '7','8','9',...
       ,..., 'A', 'B', 'C', 'D', 'E', 'F' ,..., 'a', 'b', 'c',...);
  type severity_level is (note, warning, error, failure);
-- Implementation dependent definitions
  type integer is range -(2**31-1) to (2**31-1);
  type real is range -1.0e38 to 1.0e38;
  type time is range 0 to 1e20
       units fs; ps=1000fs; ns=1000ps; us=1000ns; ms=1000us;
             sec=1000ms; min=60sec; hr=60min;
  end units time;
  .../...
  function NOW return TIME;
  subtype natural is integer range 0 to integer'high;
  subtype positive is integer range 1 to integer'high;

  'high is an attribute of data-types and refers to the highest literal value of related data-types (integer in this case)

  type string is array (positive range <>) of character;
  type bit_vector is array (natural range <>) of bit;
  .../...
End standard;
```

A.2.4 Operators

Operators are specific reserved words or symbols which identify different operations that can be performed with objects and literals from different data types. Some of them are intrinsic to the language (e.g., adding integers or reals) while others are defined in specific packages (e.g., adding bit_vectors or signed bit strings).

Operators are defined by their symbols or specific words and their operand profiles (number, order, and data type of each one); thus the same operator symbol or name can support multiple definitions by different profiles (overloaded operators). Operators can only be used with their specific profile, not with every data type.

A.2.5 VHDL Structure: Design Units

VHDL code is structured in different design units: *entity*, *architecture*, *package* (*declaration* and *body*), and *configuration*.[2] A typical single VHDL module is based on two parts or design units: one simply defines the connections between this module and other external modules, while the other part describes the behavior of the module. As shown in Fig. A.2 those two parts are the design units *entity* and *architecture*.

[2] Configuration design unit will be defined, but is not used in this text.

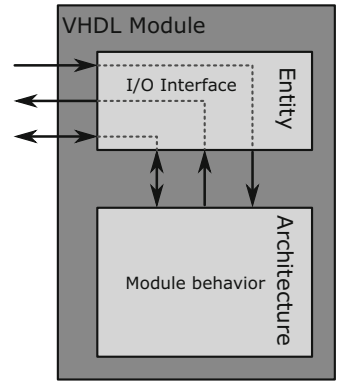


Fig. A.2 Basic VHDL module structure: entity—architecture

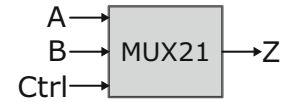
Entity: Like a "black box" with its specific name or identifier, just describing external interface for a module while hiding its internal behavior and architecture. Next text box summarizes its syntax:

```
entity <id> is -- <id> is the entity or module name
   [<generics>];-- generic parameters
   [<ports>]; -- I/O ports
   [<declarations>];-- Global declarations for the module
[begin <sentences>];-- Passive sentences
end [entity] [<id>];-- End of entity definition
```

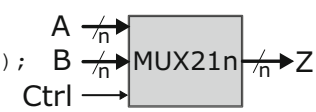
The *generic parameters* and *ports* are declared within the entity and accessible only on this entity and any associated architecture. Generics are able to pass values into modules or components giving them different qualities or capabilities (e.g., the size of input/output buses). The values of generics can differ, for the same component, from instantiation to instantiation but remain constant within the scope of each instantiation.

Ports define the input/output signal interface of the module and are used to interconnect instances of the module (component instances) with other components. Declarations and passive sentences in the entity will not be used across this overview. Refer to next examples for better understanding.

```
entity MUX21 is -- Entity name or identifier: MUX21
-- input ports: signals A, B and Ctrl of datatype "Bit".
-- output port: signals Z of datatype "Bit".
   port( A     : in  bit;
         B     : in  bit;
         Ctrl  : in  bit;
         Z     : out bit);
end MUX21;
```




```
entity MUX21n is -- Entity name or identifier: MUX21n
-- Generic parameter "n"; type: integer; default value: 8
  generic( n : integer := 8);
-- input ports: buses A, B; datatype bit_vector of n-bits
--              signal Ctrl of datatype Bit.
-- output port: bus Z of datatype bit_vector of n-bits.
  port(A : in bit_vector(n-1 downto 0);
       B : in bit_vector(n-1 downto 0);
       Ctrl : in  bit;
       Z : out bit_vector(n-1 downto 0));
end MUX21
```



Architecture: This design unit details the behavior behind an *Entity* while describing it at functional, data-flow, or structural levels. An *Architecture design unit* is always linked to its related *Entity*, but an entity could have multiple architectures:

```
-- Architecture named <id> linked to entity <id_entity>
architectura <id> of <id_entity> is
   [<declarations>]; -- Declarative section
begin
   <concurrent sentences>; -- Detailed behavior description
end [architecture] [<id>];
```

In the declarative section different data types, objects (constants, variables, signals, or files), and components for this specific architecture could be declared and they will remain visible only within this architecture unit.

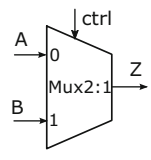
The section enclosed by "begin . . . end" is devoted to detailed behavior description by means of concurrent sentences (its writing order is not important as all of them are evaluated concurrently). This appendix will address just the most important ones: signal assignments, components instantiations, and processes. In the next examples there are three different architectures linked to the same entity (MUX21 entity).

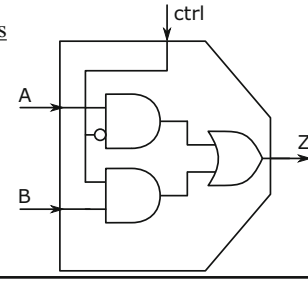
```
architecture Functional
of MUX21 is
begin
  process(A, B, Ctrl)
  begin
    if Ctrl = '0' then
      Z <= A;
    else
      Z <= B;
    end if;
  end process;
end Functional;
```

```
architecture DataFlow of
MUX21 is
 signal Ctrl_n,N1,N2:bit;
begin
 N1     <= Ctrl_n and a;
 Z      <= (N1 or N2);
 N2     <= Ctrl and b;
 Ctrl_n <= not Ctrl;
end DataFlow;
```

```
architecture structural of MUX21
is
  signal Ctrl_n, As, Bs : bit;
  component INV
    port( Y : in  bit;
          Z : out bit);
  end component;
  component AND2
    port( X, Y : in  bit;
          Z    : out bit);
  end component;
  component OR2
    port( X, Y : in  bit;
          Z    : out bit);
  end component;
begin
 U0: INV  port map (Ctrl, Ctrl_n);
 U1: AND2 port map (Ctrl_n,A,As);
 U2: AND2 port map (Ctrl, B, Bs);
 U3: OR2  port map (As, Bs, Z);
end structural;
```

Modelled MUX21 block in previous examples





Comment A.2
In any of the three previous architectures, if they are linked to a MUX21n entity instead of MUX21, the primary signals A, B, and Z will become n-bit buses.

Package: This design unit allows VHDL code reuse across different VHDL modules and projects. Within a package we can define new data types and subtypes, constants, functions, procedures, and component declarations. The package must have a *Package declaration* and may have a *Package Body* linked to the previous one by means of using the same package identifier.

Any declaration must appear in the *Package*, while the detailed definitions of functions or procedures must be done in the *Package Body*. Constants could be initialized in any one of them.

```
package <identifier>
  [<declarations>];
end [package] [<identifier>]

package body <identifier>
  [<Assignments and Detailed definitions>];
end [package body] [<identifier>]
```

Designers can define their own packages; this is a way to define the boundaries of data types, functions, and naming convention within a project. Nevertheless there are some key packages that are already defined and standardized: (1) STANDARD (partially defined above) and TEXTIO (definition of needed file management data types and functions), both of them already defined by the VHDL language itself; (2) Std_logic_1164 and Std_logic_arith which define a multivalued logic and its related set of conversion functions and arithmetic-logic operations—and both of them are under the IEEE committee's standardization management. The Std_logic_1164 package has been defined to accurately simulate digital systems based on a multivalued logic (std_logic) with a set of nine different logical values.

Std_logic_1164 multivalued logic values	
Value character	Meaning
"U"	Uninitialized
"X"	Strong drive, unknown logic value
"0"	Strong drive, logic zero
"1"	Strong drive, logic one
"Z"	High impedance
"W"	Weak drive, unknown logic value
"L"	Weak drive, logic zero
"H"	Weak drive, logic one
"-"	Don't care

Packages are either predefined or developed by the designer and compiled into libraries that must be referenced in order to get access to the related packages. This is done by means of the sentence "use" identifying the library, the package name, and the specific identifier to be used from the package or simply the clause *all* to get access to the whole package definitions. Libraries and use clauses are written at the beginning of VHDL source files to make those packages accessible from any design unit in the file:

```
use <library>.<package name>.[<identifier> | all];
```

This section ends with two package examples. The package *main_parameters* has been defined to be used for the simple microprocessor modelled throughout this book. In this package just constants are defined and initialized in the package declaration; thus, package body has not been used in this case (see Chap. 5 for its definition and usage).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
package main_parameters is
constant m: natural := 8; -- m-bit processor
-- Value '0' in m-bits
constant zero: std_logic_vector(m-1 downto 0):=
conv_std_logic_vector(0, m);
-- Value '1' in m-bits
constant one: std_logic_vector(m-1 downto 0) :=
conv_std_logic_vector(1, m);
-- Our simple processor instruction set codes definition
constant ASSIGN_VALUE: std_logic_vector(3 downto 0) :=
"0000";
constant DATA_INPUT: std_logic_vector(3 downto 0):="0010";
…/…
constant OPERATION_ADD: std_logic_vector(3 downto 0) :=
"0100";
…/…
constant JUMP: std_logic_vector(3 downto 0):="1110";
constant JUMP_POS: std_logic_vector(3 downto 0):="1100";
constant JUMP_NEG: std_logic_vector(3 downto 0):="1101";
end main_parameters;
```

Standard packages from IEEE libraries

Conversion functions from IEEE std. Pack.

The second package *example* is a user-defined package with both package declaration and package body definitions, including a data type, a constant, and a function. The constant value and the detailed function code are defined in the related package body:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package example is
 type redlightcolors: (red, yellow, green);
 constant defaultcolor: redlightcolors;
 function color_to_int (variable color: in redlightcolors)
          return integer;
end example;

Package body example is
 constant defaultcolor: redlightcolors := yellow;
 function color_to_int (variable color: in redlightcolors)
          return integer is
  variable colnum: integer; -- local variable
  begin
   case color is
     when red then colnum := 0; -- value '0' is red
     when yellow then colnum := 1; -- value '1' is yellow
     when green then colnum := 2; -- value '2' is green
```

(continued)

```
  end case;
  return colnum;-- returned value
 end;
end example;
```

The ***Configuration*** declaration design unit specifies different bounds: architectures to entities or an entity-architecture to a component. These bound definitions are used in further simulation and synthesis steps.

If there are multiple architectures for one entity, the configuration selects which architecture must be bound to the entity. This way the designer can evaluate different architectures through simulation and synthesis processes just changing the configuration unit.

For architectures using components, the configuration identifies a specific entity-architecture to be bound to a specific component or component instantiation. Entity-architecture assigned to components can be exchanged if they are port compatible with the component definition. This allows for a component-based architecture, the evaluation of different entity-architectures mapped into the different components of such architecture.

Configuration declarations are always optional and in their absence the VHDL specifies a set of rules for a default configuration. As an example, in the case of multiple architectures for an entity, the last compiled architecture will be bound to the entity for simulation and synthesis purposes by default. As this design unit is not used in this book, we will not consider it in more depth.

A.3 Concurrent and Sequential Language for Hardware Modelling

Hardware behavior is inherently concurrent; thus, we need to model such a concurrency along time. The VHDL language has resources to address both time and concurrency combined with sequential behavior modelling.

Before going into different sentences let's review the general VHDL model structure summarized in Fig. A.3.

Looking at this figure we realize that in the architecture description domain only concurrent sentences are allowed; in the process description domain only sequential sentences are possible, the process itself being one of the key concurrent sentences.

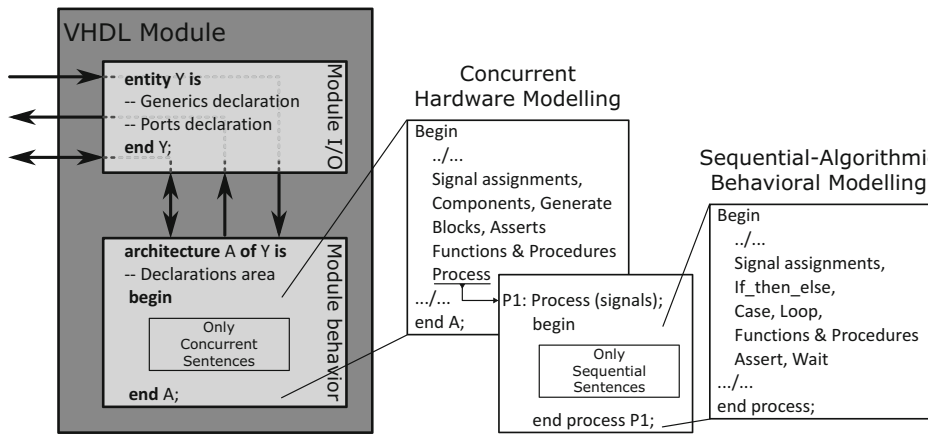


Fig. A.3 General VHDL module structure: concurrent and sequential domains

Concurrent sentences are evaluated simultaneously; thus the writing order is not at all relevant. Instead, sequential sentences are evaluated as such, following the order in which they have been written; in this case the order of sentences is important to determine the result.

Generally speaking, it can be said that when we use concurrent VHDL statements we are doing concurrent hardware modelling, while when we use sequential statements we perform an algorithmic behavioral modelling. In any case, both domains can be combined in VHDL for modelling, simulation, and synthesis purposes.

Figure A.3 also identifies the main sentences, concurrent and sequential, that we will address in this introduction to VHDL, the process being one of the most important concurrent sentences that links both worlds within VHDL. In fact, any concurrent sentence will be translated into its equivalent process before addressing any simulation or synthesis steps, which are simply managing lots of concurrent processes.

A.3.1 Sequential Sentences

Sequential sentences can be written in Processes, Functions, and Procedures. Now we will review a small but relevant selection of those sentences.

Variable assignment sentence has the same meaning and behavior as in software languages. The variable changes to the new value immediately after the assignment. Below you can find the syntax where the differentiating symbol is ":=" and the final expression value must belong to the variable data type:

```
Syntax:    [label:] <variable name> := <expression>;
Examples: Var := '0';   C := my_function(4,adrbus) + A;
          Vector := "00011100"; B := my_function(3,databus)
          string := "Message is: "; A := B + C;
```

Signal assignment sentence is used to project new value-time events (value v_i will become active at time t_i) onto the related signal driver (review Fig. A.1). The new projected events are not immediately installed into related signal drivers; only when the process is suspended the new events are analyzed, resolved, and updated into the corresponding signal drivers.

Syntax and a few examples follow. In this case the assignment symbol is "<=" and the result of the expression must have the same size (number of bits) and data type as the assigned signal. The clause *after* refers to a delay which is applied to the related event before being effective in the assigned signal. When *after* is missing, the applied delay is 0 ns, which is known as *δ-delay* in VHDL (see Sect. A.4). In the next example the assignment to signal *X* has no delay, so it is a *δ-delay*:

```
Syntax:
[label:] <signal_name> <= [delay_type] <expression> {after <delay>};
Examples:
Reset <= '1','0'after 10ns; --10ns. Pulse on Reset signal
X <= (A or B)and C; --Boolean expression result to X sig.
-- Waveform (values-times) projected to signal Y
Y <= '0', '1' after 10 ns, '0' after 15 ns, '1' after 20;
```

Wait sentence is used to synchronize processes in VHDL. The processes communicate through signals; the signals are updated along the simulation cycle only when all the processes at the current simulation time have been stopped either as a result of reaching a wait sentence or waiting for events in the sensitivity signal list (see concurrent *Process* sentence). Then the signal drivers are updated

with the latest projected events, and immediately after that the simulation time advances to the closest event in time in any of the circuit signal drivers to evaluate the effects of events at that time.

The wait sentence specifies where the process is suspended and which are the conditions to resume its execution. Formal syntax and a few examples follow:

```
[label:] wait [on <signal> {, ...}]
              [until <boolean_expresion>]
              [for <time_expresion>];
```

Process is suspended forever

```
process
begin
  <sequential sentences>
  wait;
end process;
```

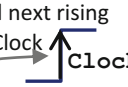
Suspended during 10 ns

```
process
begin
  Clock <= not Clock;
  wait for 10 ns;
end process;
```

Suspended waiting for any event on signals 'a' or 'b'

```
process
begin
  c <= a and b;
  wait on  a, b;
end process;
```

Suspended until next rising edge on signal Clock




```
process
begin
  q <= d;
  wait until Clock = '1';
end process;
```

Most processes only have one wait sentence or its equivalent sensitivity list, but more than one wait per process is possible (see test-bench concept and examples).

If_then_else sentence is quite similar to its software equivalent statement, but oriented to hardware modelling. Syntax and a few examples follow.

The *Mux* (already described previously) and *Tristate* are combinational processes, that is, any event in the inputs can cause an event at the outputs. In the case of *Tristate* the output (*Sout*) follows the input (*Sinp*) when *enable* = "*1*", but the output becomes *"Z"* (high impedance or disconnected) when *enable* = "*0*":

```
[label:]if <condition> then <sequential sentences>
        {elsif <condition> then <sequential sentences>}
        [else <sequential sentences>]
        {end if;}
       end if [label];
```

```
Mux: Process (A,B,Ctrl)
Begin
  if Ctrl = '0' then
     Z <= A;
  else
     Z <= B;
  end if;
end process Mux;
```

```
Tristate: Process(enable,sinp)
begin
  if Enable ='1' then
    Sout <= Sinp;
  else
    Sout <= 'Z';
  end if;
end process Tristate;
```

```
Latch: Process (Load,D)
Begin
  if Load = '1' then
     Q <= D;
  end if;
end process Latch;
```

```
FlipFlop: Process (rst,clk)
begin
 if rst = '1' then Q <= '0';
  elsif (clk'event and clk='1')
    then Q <= D;
 end if; end if;
end process FlipFlop;
```

The *Latch* and *FlipFlop* processes model devices able to store binary information, and both of them contain one incompletely specified *if-then-else* sentence: in the latch model there is no *else* clause and in the flip-flop model there is no inner *else* clause. In both cases when the clause *else* becomes true, no action is specified on the output (*Q*); therefore such signal keeps the previous value; this requires a memory element. The above codes model such kind of memory devices.

In the above latch case, *Q* will accept any new value on *D* while *Load* = "*1*", but *Q* will keep the latest stored value when *Load* = "*0*".

In the other example, flip-flop reacts only to events in signals *rst* and *clk*. At any time, if *rst* = "*1*" then *Q* = "*0*". When *rst* = "*0*" (not active) then *clk* takes the control, and at the rising edge of this *clk* signal (condition *clk'event*[3] *and clk* = "*1*"), then, and only then, the *Q* output signal takes the value from *D* at this precise instant of time and stores this value till next *rst* = "*1*" or rising edge in *clk*.

Case sentence allows selecting different actions depending on non-overlapping choices or conditions for a specific expression. Syntax and a few examples follow:

```
[label:] case <expresion> is
           when <choices> => <sequential sentences>;
         {[when <choices> => <sequential sentences>;]}
          [when others => <sequential sentences>;]
        end case [label];
```

```
CaseALU: process (Op1, Op2, Operation)
begin                          Operation enumerated datatype: (add, subs, andL, orL)
  case Operation is
    when add  => Result <= Op1  +  Op2;
    when subs => Result <= Op1  -  Op2;
    when andL => Result <= Op1 and Op2;
    when orL  => Result <= Op1  or Op2;
  end case;
end process CaseALU;
```

Op1, Op2 and Result signals share same datatype and size

```
CasExample: process (Valin)
begin
  case Valin is
    when 0         => Res := 5;
    when 1 | 2 | 8 => Res := Valin;
    when 3 to 7    => Res := Valin + 5;
    when others    => Res := 0;
  end case;
  Valout <= Res after 10ns;
end process CasExample;
```

The *when* <*choices*> clause can refer to explicit <*expression*> data type values, as in the previous *CaseALU* example, or to alternative values ($v_i|v_j|v_k$), range of values (v_i to v_k), or any other value not yet evaluated (*others*), as in *CasExample*. The list of *when* <*choices*> clauses must not overlap (see previous example). The *others* choice is useful to avoid incompletely specified case sentences, which will model a latch structure for all the non-specified choices.

This sentence has been intensively used in Chap. 5 to model different conditional selections to build up our simple microprocessor.

[3] event is a signal attribute that is "true" only when an event arrives to related signal. VHDL attributes are not explicitly addressed in this text; just used and commented when needed.

Comment A.3
Any conditional assignment or sentence where not all the condition possibilities have been addressed, explicitly or with a default value (using clause "others" or assigning a value to related signal just before the conditional assignment or sentence), will infer a latch structure.

Loop sentence collects a set of sequential statements which are repeated according to different loop types and iteration conditions. Syntax follows:

```
[label:] [while <boolean_condition> | for <repetition_control>]
         loop
           {<sequential sentences>}
         end loop [label];
```

For *while-loop* type the sentences in the loop are repeated until the *<boolean condition>* becomes *false*, and then continues with the next sentence after the *end loop*.

In the *for-loop* type the number of iterations is controlled by the range of values indicated by *<repetition_control>*.

```
Count16: process
begin
  Cont <= 0;
  loop
    wait until Clock='1';
    Cont <= (Cont+1) mod 16;
  end loop;
end process;
```

```
Count16: process
begin
  Cont <= 0;
  wait until Clock='1';
    while Cont < 15 loop
      Cont <= Cont + 1;
      wait until Clock='1'
    end loop;
end process;
```

Finally an infinite loop is also possible if you avoid the above while/for-loop types. Such a loop will never stop. The previous two text boxes model the same behavior: a counter modulo 16 using an infinite loop and a *while* controlled loop.

The following *for-loop* type example corresponds to a generic n-bits parallel adder model with both, entity and architecture design units. Before entering the loop we assign signal Cin to variable C(0). Then each iteration along the loop computes the addition of two bits, X(I) and Y(I), plus carry in, C(I), and generates two outputs: signal Z(I) and variable C(I + 1). Once the loop finished the variable C(n) is assigned to the primary output signal carry out (Cout):

```
entity ParallelAdder is
 generic (n : natural :=4);
 port ( X, Y : in std_logic_vector(n-1 downto 0);
       Cin : in std_logic;
       Z : out std_logic_vector(n-1 downto 0);
       Cout : out std_logic);
End ParallelAdder;
```

```
achitecture Functional of ParallelAdder is
begin
  process (X, Y, Cin);
```

(continued)

```
  variable C : std_logic_vector(n downto 0);
  variable tmp : std_logic;
  variable I : integer;
  begin
    C(0) := Cin;
    for I in 0 to n-1 loop
     tmp  := X(I) xor Y(I);
     Z(I)  <= tmp xor C(I);
     C(I+1) := (tmp and C(I)) or (X(I) and Y(I));
   end loop;
   Cout <= C(n);
 end process;
end Functional;
```

Comment A.4

From synthesis point of view any loop iteration will infer all the needed hardware for its implementation. The number of iterations must be known at VHDL source code level, thus independent of code execution.

Nested loops are possible; in such a case it is convenient to use the optional [*Label*] clause with care in order to correctly break the loop execution when specified conditions are reached. In addition to the normal end of a loop when conditions/iterations finish, there are two specific sentences to break a loop and those are *Exit* and *Next*, with syntax as follows:

```
[label:] exit [label_loop] [when <Boolean-condition>];
[label:] next [label_loop] [when <Boolean-condition>];
```

Exit sentence ends the associated [*label_loop*] iterations when its related <*boolean-condition*> is true; the execution continues to the next sentence after the loop.

The ***next*** sentence, on the other hand, ends the current [*label_loop*] iteration when its related <*boolean-condition*> is true; then the next iteration starts. This allows to skip specific loop iterations.

The previous *Count16* examples have been slightly modified to show how *exit* and *next* sentences work. In the case of the *Count16skip10* process the counting loop skips the value "*10*" in the signal *Cont*. The *Count16isCount10* process simply exits the counting at value *Cont* = *10* and restarts again at *Cont* = *0*:

```
Count16skip10: process
begin
  Cont <= 0;
  loop
   wait until Clock='1';
   next when (Cont = 9);
   Cont <= (Cont+1) mod 16;
  end loop;
end process;
```

```
Count16isCount10: process
begin
  Cont <= 0;
  wait until Clock='1';
    while Cont < 15 loop
     Cont <= Cont + 1;
     wait until Clock='1';
     exit when (Cont=10);
    end loop;
end process;
```

Function declaration:

```
function <name> [(<parameters list>)] return <data_type>;
```

Function definition:

```
function <name> [(<parameterslist>)] return <data_type> is
 {<declarative part>}
begin
 {<sequential sentences>}
end [function] [<name>];
```

Example of function declaration:

```
function bv2int (bs: bit_vector(7 downto 0))
          return integer;
```

Example of function usage:

```
Var := base + bv2int(adrBus(15 downto 8));
Sig <= base + bv2int(adrBus(7 downto 0));
```

Functions and Procedures have the same meaning and function as their counterparts in software:

- Functions are pieces of code performing a specific computation, which depending on input parameters returns a value by means of ***return*** sentence.
- A Procedure models a specific behavior or computation, which takes the input parameter values and returns results through the output parameters.

Both of them are usually defined within packages to facilitate their reusability in any VHDL code by means of *use* sentence. Syntax and a few declaration and usage examples are given in the previous (function) and next (procedure) text boxes.

Procedure declaration:

```
procedure <name> [(<parameters list>)];
```

Procedure definition:

```
function <name> [(<parameterslist>)] is
  {<declarative part>}
begin
  {<sequential sentences>}⟵ [label:] return [expresion];
end [procedure] [<name>];
```

Example of procedure declaration:

```
procedure bv2int (bs: in bit_vector(7 downto 0);
                  x: out integer );
```

Example of procedure usage:

```
bv2int(adrBus(15 downto 8); Var);
Var := base + Var;
```

Assert sentence checks the <*boolean expression*> and reports notification messages of different error severities. When the <*boolean expression*> is *FALSE* then the <*string of characters*> is printed or displayed and the simulator takes the specific actions associated to the severity level specified. Syntax and a few examples follow:

```
[label:] assert <boolean expression>
            [report <string of characters>]
            [severity (note | warning | error | failure);
```

```
assert not(addr < X"00001000" or addr > X"0000FFFF")
       report "Address in range" severity note;
assert (J /= C) report "J = C" severity note;
```

The ***sentence report*** has been included in VHDL for notification purposes—not to check any Boolean expression—with the following syntax and related example:

```
[label:] [report < string of characters >]
            [severity (note | warning | error | failure);
Example:
report "Check point 13"; -- is fully equivalent to ...
assert FALSE "Check point 13" severity note;
```

A.3.2 Concurrent Sentences

Remember from Fig. A.3 that all the sentences within *architecture design units* are concurrent. All concurrent sentences are evaluated simultaneously; thus, their written order is not at all relevant. Different orderings for the same set of concurrent sentences must produce the same simulation and synthesis results.

Concurrent sentences can be used on different modules: *Entity* (in the part devoted to passive sentences), *Block* (groups of concurrent sentences inside any architecture for code organization purposes), and *Architectures*. In this introduction we will just address their usage inside architectures, since it is the most common and important design unit to model concurrency.

Now we will review the most relevant subset of those sentences. We already know that any concurrent sentence is translated, before simulation or synthesis, into its equivalent *Process*. Thus, simulation and synthesis work with lots of concurrent processes. We will show examples and their equivalent processes for some of the concurrent sentences.

The ***Process sentence*** is a concurrent sentence that collects sequential statements to define a specific behavior. The communication among processes is done by means of signals. Different events or conditions on these communicating signals will activate or resume the process evaluation once stopped by a *wait* sentence.

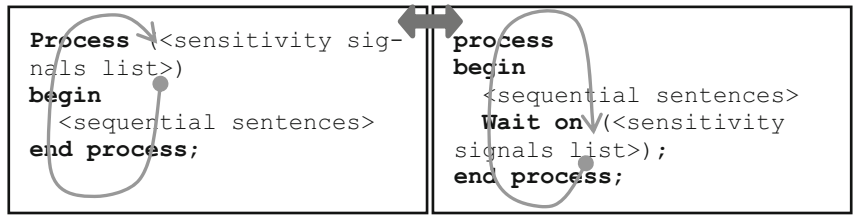
The next text box shows the process syntax. The *<sensitivity signals list>* after the *process* reserved word is optional, and it means that any event on any of these signals will activate the process reevaluation at the time the related event occurred. In the process sentence it is also possible to include local declarations of VHDL objects, which will be visible only within the framework of the related process. Finally in the *begin. . .end process* section the process behavior is described by means of sequential statements.

```
[<label>:] process [(<signal> ,{<signal>, ...})] [is]
  [<local declarations>]
begin
  <sequential sentences>
end process [<label>];
```

Sensitivity signals list: events on any of them will reactivate the process execution

The *begin. . .end* section is an infinite loop that must at least contain either a *wait* sentence inside or a *<sensitivity signals list>* at the beginning of the process. A process with a sensitivity signal list at

the beginning is equivalent to a process with just a wait statement at the end as you can see in the next text boxes.



Signal assignment sentences: In the concurrent world the same sequential signal assignment sentence syntax and rules could be used (review related section), but now it works as concurrent sentence. Nevertheless in concurrent VHDL there are other possibilities for signal assignment that follow the same rules but with wider functionalities, as explained in the next paragraphs.

Conditional assignment (***when/else***) is used when different values could be assigned depending on different Boolean conditions, which are explored on a given priority: the condition tested earlier (closer to the assignment symbol <=) is evaluated before subsequent conditional assignments in the same sentence. As Boolean expressions can be completely independent from one another, there is the possibility of coding overlapped conditions. In such a case the abovementioned order of testing the Boolean expressions determines the priority of assignment. The following text box shows the syntax of this sentence:

```
[<label>:] <signal> <= [delay_type]
  {<expression|waveform> when <boolean expression> else}
  <expression|waveform> [when <boolean expression>]
  [[<expression|waveform> | unaffected] when others];
```

In the case that the final *else* is missing or that the final clause is *unaffected when others* in the chained list of *when/else* clauses, the target signal must keep its previous value; thus a latch is modelled and will be inferred by synthesis tools. Using a final *else* clause instead of *unaffected when others* will avoid accidental latch inference.

As you can see in the next examples this sentence behaves like an *if_then_else* with nesting possibilities. In fact, the equivalent process for this concurrent sentence is modelled with such a sequential *if_then_else* sentence.

The second example below shows a more complex conditional list of assignments which corresponds to a sequence of nested *if_then_else*, and its equivalent process is modelled consequently.

Concurrent sentence:

```
S <= A when Sel = '0' else B;
-- Sel is one bit signal
```

Equivalent process:

```
process (Sel, A, B)
begin
  if Sel = '0' then
    S <= A;
  else
    S <= B;
  end if;
end process;
```

Concurrent sentence:

```
S <= E1 when Sel2 = "00" else
     E2 when Sel2 = "11" else
     unaffected when others;
-- Sel is two bits signal
```

Due to unaffected clause a latch will be inferred on signal S to keep its previous value when Sel2 value is neither "00" nor "11".

Equivalent process:

```
process (Sel2, E1, E2)
begin
  if Sel2 = "00" then
    S <= E1;
  elsif Sel2 = "11" then
    S <= E2;
  else
    null;
  end if;
end process;
```

Selective assignment (with/select) is another type of conditional assignment. Instead of allowing the analysis of completely different conditions in the Boolean expressions as in the previous sentence (when/else), the with/select statements make the assignment selection based on the value of only one signal or expression (see the next text box for the related syntax):

```
[<label>] with <expression> select
 <signal> <= [delay_type]
             {<expression|waveform> when <value>,}
              <expression|waveform> when <value>,
             [<expression|waveform> when others];
```

In this type of selective assignments overlapping conditions are forbidden; this sentence is equivalent to a *Case* statement. If the selection list doesn't covers every possibility of the *<expression>* then a default value can be assigned using the *when others* clause to avoid undesired latch inference.

The next example shows a simple usage of this concurrent selective assignment sentence and the equivalent concurrent process, which is based on a sequential *Case* statement:

Concurrent sentence:

```
with Operation select
  Result <= Op1  +  Op2 when add,
            Op1  -  Op2 when subs,
            Op1 and Op2 when andL,
            Op1  or Or2 when orL;
```

Operation enumerated datatype is: (add, subs, andL, orL). Op1, Op2 and Result signals share same datatype and size.

Equivalent process:

```
process (Op1, Op2, Operation)
begin
  case Operation is
    when add  => Result <= Op1  +  Op2;
    when subs => Result <= Op1  -  Op2;
    when andL => Result <= Op1 and Op2;
    when orL  => Result <= Op1  or Op2;
  end case;
end process;
```

Components are modules that have been defined (entity-architecture) somewhere and declared within a package or in the declarative part of the architecture willing to use them. Now we will deal with hierarchy and instantiation concepts by using components. A component is a module defined by its *Entity-Architecture*. Declaring a component is simply identifying its name, related generic parameters (if any), and input/output ports to allow multiple and different references to it with different parameter values and port connections. We use a component by means of a reference or instantiation to it with specific generic parameters and port connections:

Component declaration syntax:

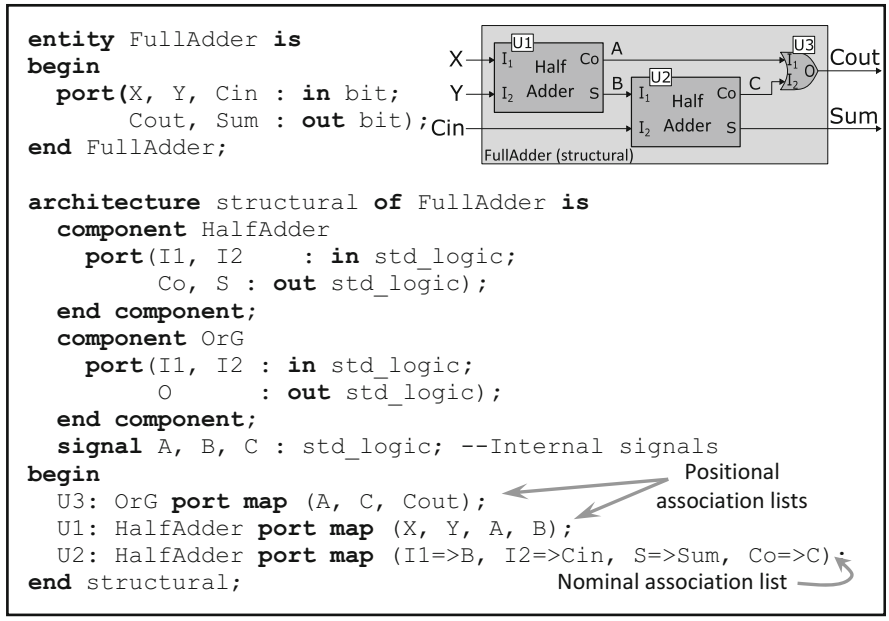
```
component <idname> [is]
    [generic (<generic parameters list>);]
    [port (<ports list>);]
end [component] [<idname>];
```

Component instantiation syntax:

```
<label>: <idname>
    [generic map (<parameters association list>);]
    [port map (<ports association list>);]
```

The component declaration is like a template that must match the entity of the related module. The component instantiation takes one copy of the declaration template and customizes it with parameters and ports for its usage and connection to a specific environment. The *<label>* in the instantiation is important to distinguish between different instances of the same component throughout simulation, synthesis, or other analysis tools.

Both parameter and port association lists must be assigned or mapped into actual parameters and signals in the architecture instantiating the component. Such an assignment could be done by position or by name. In a positional assignment the order is important, as each actual parameter/signal listed in the generic/port map is connected to the formal parameter/port in the same position in component declaration. In an assignment by name the order is irrelevant, as each actual parameter/port is explicitly named along with the parameter/signal to which it is connected. The main advantages of nominal assignment are its readability, clarity, and maintainability. Next example shows the main characteristics of component usage.



This structural architecture style based on component instantiation is like connecting components as a netlist. On top of the example you can see the related graphic schematic view of such an architecture description.

In the above example we do not really know which are the entity-architecture modules associated to each component. We can add the following configuration sentences after a component declaration to perform such an association between components {*HalfAdder, OrG*} and modules {*HAcell, Orcell*} from *LibCells* library using the so-called *structural* and *functional* architectures, respectively. In this case we have done an explicit configuration inside the architecture, but there are other possibilities when using the configuration design unit:

```
for all: HalfAdder use entity LibCells.HAcell(structural);
for U3: OrG use entity LibCells.Orcell(functional);
```

In VHDL it is also possible to avoid component declaration and configuration by doing direct instances or references to related entity-architecture modules. Using this strategy in the above example we can get the following architecture:

```
architecture structural of FullAdder is
 signal A, B, C : std_logic; --Internal signals
begin
  U3: entity LibCells.Orcell(functional)
                    port map (A, C, Cout);
  U1: entity LibCells.HAcell(structural)
                    port map (X, Y, A, B);
  U2: entity LibCells.HAcell(structural)
                port map (I1=>B, I2=>Cin, S=>Sum, Co=>C);
end structural;
```

Generate sentence contains further concurrent statements that are to be replicated under controlled criteria (conditions or repetitions). As you can see in the syntax, it looks similar to a *for_loop* and indeed behaves like and serves a similar purpose, but in a concurrent code. In this sense all the iterations within the generate loop will produce all the needed hardware; the number of iterations must be known at VHDL source code level. Thus, they cannot depend on code execution:

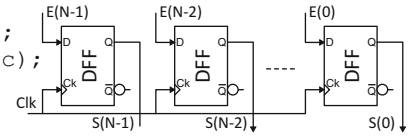
```
<label>: {[for <range specification> | if <condition> ]}
generate
 {<concurrent sentences>}
end generate;
```

The <*label*> is required to identify each specific generated structure. There are two forms for the generate statement, conditional and repetitive (loop), and both can be combined. Any concurrent statement is accepted within the body of generate; however the most common are component instantiations.

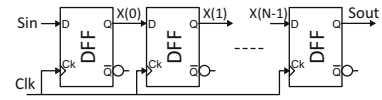
Often the generate sentence is used in architectures working with generic parameters, as such parameters are used to control the generate conditions (either form). This way a module can be quickly adjusted to fit into different environments or applications (i.e., number of bits or operational range of the module).

The following examples deal with generic parameters and regular module generation. The first example models an N-bit parallel input/output register using a *for* repetitive clause. The second models an N-bit shift register combining *for* clause to generate N-bits and *if* clauses in order to identify and distinguish the inner inter-bit connections from the serial input and serial output connections. In both cases the related schematic with flip-flops is shown.

```
entity Register is
  generic (N: positive);
  port( Clk : in  std_logic;
        E   : in  std_logic_vector(N-1 downto 0);
        S    out std_logic_vector(N-1 downto 0));
end Register;
architecture structural of  Register is
  component DFF
    port (Clk, E : in std_logic;
          S      : out std_logic);
  end component;
  variable I : integer;
begin
  GenReg: for I in N-1 downto 0 generate
    Reg: DFF port map(Clk, E(I), S(I));
  end generate;
end structural;
```



```
entity ShiftReg is
  generic (N: positive);
  port( Clk, SIn : in bit ;
        SOut     : out bit);
end ShiftReg;
architecture structural of  ShiftReg is
  component DFF
    port (Clk, E : in  bit;
          S      : out bit);
  end component;
  signal X : bit_vector(0 to N-2);
  variable I : integer;
begin
 GenShReg: for I in 0 to N-1 generate
  G1 : if (I=0) generate
        CIzq: DFF port map(Clk, SIn, X(I)); end generate;
  G2 : if (I>0) and (I<N-1) generate
      CCen: DFF port map(Clk, X(I-1), X(I)); end generate;
  G3 : if (I=N-1) generate
      CDer: DFF port map(Clk, X(I-1), SOut); end generate;
 end generate;
end structural;
```



Functions, Procedures, and Assert concurrent sentences have the same syntax and behavior as their equivalent sequential sentences, but used in the concurrent world. Thus, any additional comment will be referring to in this sense.

Concurrent Processes versus Procedures. A *Process* can be modelled by means of its equivalent *Procedure* simply by moving all the input/output process signals (including signal in the sensitivity list) to the input/output procedure parameters. The next example shows such equivalence.

```
Signal rst, ck, d, q: std_logic;
…/…
FFD: process (rst, clk);
begin
 if rst = '1' then
    Q <= '0';
 elsif (clk'event and clk='1') then
    Q <= D;
 end if; end if;
end process;

procedure FFD (signal rst, clk, d: in std_logic;
               signal q: out std_logic);
begin
 if rst = '1' then
    Q <= '0';
 elsif (clk'event and clk='1') then
    Q <= D;
 end if; end if;
end procedure;
```

Formal parameters to be assigned with the related actual parameters of each procedure instantiation.

The *Process* is defined and used in the same place. For multiple uses you need to cut and paste it or move it into a module (entity-architecture), to use it as a component to allow an easy reuse.

The *Procedure* is declared and defined within a *package*, and the clause *use* makes it accessible to be instantiated with actual parameters assigned to formal parameters; thus, it is easily reusable. The concurrent procedure call is also translated to its equivalent process before any simulation, synthesis, or analysis steps.

Comment A.5

We have finished the introduction to VHDL language; now you will be able to read and understand the examples of digital circuit models (combinational-sequential logic and finite-state machines) already presented in previous chapters.

Combinational and sequential VHDL models related to processor blocks have been introduced in Chap. 5 during the simple processor design. Finite-state machine definition and VHDL coding are presented in Sect. 4.8, while different examples are shown in Sect. 4.9.

A.4 VHDL Simulation and Test-Benches

A.4.1 VHDL-Based Design Flow

Until now we have seen the fundamentals of VHDL language and several examples on how to use it for simple hardware modelling. Now we will briefly review how to use this language in a VHDL-based design flow, as shown in Fig. A.4.

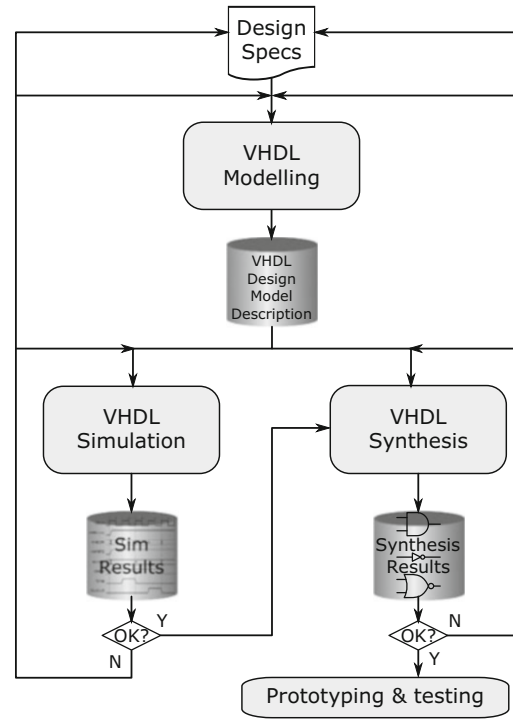


Fig. A.4 General VHDL-based design flow

Starting from design specifications, the first step is to develop a VHDL model for a target hardware that fulfills the specifications for both functionality and performance. The VHDL model produced in this initial ***modelling*** phase will be the starting point for further simulation and synthesis steps.

In order to ensure that the model fulfills the requested specifications we need to validate it by means of several refinement iterations of modelling, simulation and analysis steps.

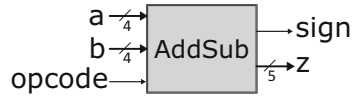
Once the quality of the model is assured by simulation we can address the next step: the hardware design or synthesis to get the model materialized in the target technology. Once again, this phase may require several improvement iterations.

Writing models with VHDL is the one and only way to learn the language. To do so we need to verify that all models do what they are intended to do by means of test-bench-based simulations.

Modelling with VHDL has been addressed through small examples in this appendix and in other larger examples throughout Chaps. 4, 5, and 6 of this book. Now we will address the basic concepts around simulation by means of a simple example. First of all, we are going to roughly define the specifications of our first target example:

Specifications: develop a hardware module able to perform additions and subtractions depending on the value of input signal "opcode {add, sub}". The operands are two non-negative 4-bit numbers, "a" and "b" (std_logic_vector(3 downto 0)). The outputs provided by this module must be:

- The result on "z" (std_logic_vector(4 downto 0)) of related operation.
- The sign of the result on signal "sign" (0: positive; 1: negative).



VHDL modelling: Based on the above specifications we have developed the next VHDL code, representing a model for a potential solution. Note that in order to have the same data type operand size when assigning values to output "z" we extend by one bit the operands on the right-hand signal assignment. At this point you must be ready and able to read and understand it:

```
package my_package is
 type operation is (add, sub);
end my_package;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.my_package.all;
entity AddSub is
    Port (a, b: in std_logic_vector(3 downto 0);
          opcode : in operation;
          sign : out std_logic;
          z : out std_logic_vector(4 downto 0));
End entity AddSub;

Architecture Functional of AddSub is
Begin
```

(continued)

```
  Operation: process (a, b, opcode)
    variable x, y : std_logic_vector(3 downto 0);
    variable s : std_logic;
 begin
  if a >= b then x := a; y := b; s := '0';
           else x := b; y := a; s := '1'; end if;
  case opcode is
    when add then z <= ('0'&x) + ('0'&y); sign <= '0';
    when sub then z <= ('0'&x) - ('0'&y); sign <= s;
    --&: concatenation operation for bit or character strings
   end case;
 end process Operation;
End Architecture Functional;
```

A.4.2 VHDL Simulation and Test-Bench

The next step in the design flow is simulation. To address it from VHDL we need to understand the fundamentals of how it works based on the concept of test-bench (TB). In order to simulate the behavior of a target VHDL module, the module-under-test (MuT), we need to define and apply stimuli to the inputs of such a module, perform a simulation, and study how the MuT reacts and behaves. Such a flow is shown in Fig. A.5.

The simplest ***test-bench*** is just another VHDL module prepared to connect with the MuT in order to generate and apply stimuli to its input signals during the simulation of both modules together. Additionally, a test-bench can also analyze the results from MuT to automatically detect mistakes and malfunctions. To do so, full VHDL capabilities are available for test-bench development in order to reach different levels of interaction, smartness, and complexity. Figure A.6 (left-hand side) shows a schematic view of this concept.

Test-bench systems are self-contained, without external connections. Once the complete test-bench environment is ready we can run the simulation to observe and analyze the signals' evolution throughout time for both, primary inputs/outputs and internal signals.

In some cases we use only the stimuli generation part of the test-bench without any feedback from MuT to TB. This simplified TB will be the case of our example AddSub simulation (Fig. A.6, right-hand side).

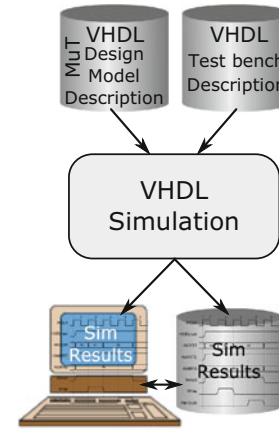


Fig. A.5 Basic simulation flow and related elements

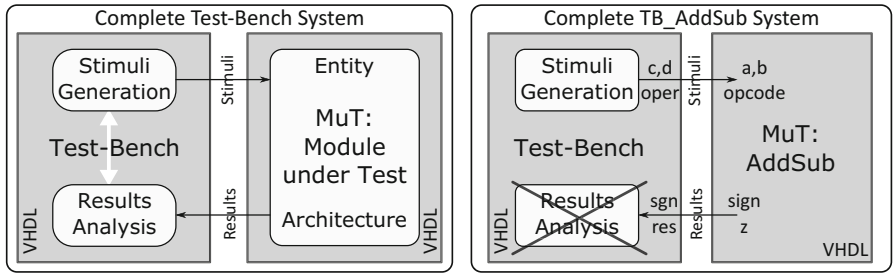


Fig. A.6 Schematic generic view of VHDL test-bench for MuT simulation (*left hand*). Simplified view of complete VHDL test-bench for AddSub module simulation (*right hand*)

According to Fig. A.6 we have prepared the following test-bench for the already modelled AddSub module (Entity-Architecture): *AddSub* (*Functional*).

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.my_package.all;
entity TB_AddSub is end TB_AddSub;

architecture Test_Bench of TB_AddSub is
   signal c, d: std_logic_vector(3 downto 0);
   signal oper: operation; signal sgn: std_logic;
   signal res: std_logic_vector(4 downto 0);
begin
  MUT: entity work.AddSub(functional)
              port map(c, d, oper, sgn, res);

stimuli: process
  begin
    c <= "0110"; d <= "0101"; oper <= add; wait for 100ns;
    oper <= sub; wait for 100 ns;
    d <= "1001"; wait for 100 ns;
    oper <= add; wait;
  end process stimuli;
end Test_Bench;
```

Module under Test

Stimuli waveforms

Observe that this *TB_AddSub* test-bench module doesn't have any I/O port, as it is only used to generate and apply stimuli to the *AddSub*(*Functional*) module in order to perform its simulation.

In the declarative part of *Test_Bench* architecture we define the signals to connect to MuT. The architecture is described by two concurrent statements: (1) the instance or reference to MuT: *AddSub (Functional)*, and (2) the process *stimuli* to schedule the different waveforms on the input signals of *AddSub* module under test.

Figure A.7 shows the results of simulation once the events scheduled by *Stimuli* process have been applied to the input ports of the *AddSub* module every 100 ns until reaching the end of this process.

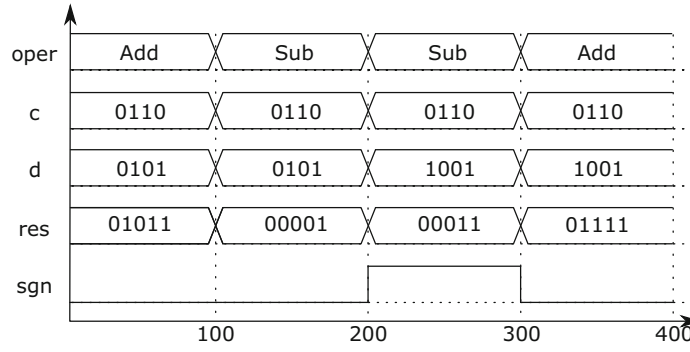


Fig. A.7 Simulation chronogram of TB_AddSub module

A.5 Summary

This appendix provides an introduction to VHDL language, including a short historical review; the basics of its lexicon, syntax, and structure; and concurrent and sequential sentences. Most of the introduced concepts are supported by specific VHDL modelling examples as well as references to other chapters in this book.

The final section has been devoted to completing this introduction with the basics (module-under-test and test-benches) for such an important topic as is the verification of the models by means of their simulation.

References

Ashenden PJ (1996) The designer's guide to VHDL, 3rd edn. 2008. Morgan Kaufmann Publishers, San Francisco
Ashenden PJ, Lewis J (2008) VHDL 2008: just the new stuff. Morgan Kaufmann Publishers, San Francisco
Kafig W (2011) VHDL 101: everything you need to know to get started. Elsevier, Burlington
Terés L, Torroja Y, Olcoz S, Villar E (1998) VHDL: Lenguaje estándar de diseño electrónico. McGraw-Hill, Germany

Appendix B: Pseudocode Guidelines for the Description of Algorithms

Mercè Rullán

The specification of digital systems by means of algorithms is a central aspect of this course and many times, in order to define algorithms, we use sequences of instructions very similar to programming language instructions. This appendix addresses those who do not have previous experience in language programming such as C, Java, Python, or others, or those who, having this experience, want to brush up their knowledge.

B.1 Algorithms and Pseudocode

An algorithm is a sequence of operations whose objective is the solution of some problem such as a complex computation or a control process.

Algorithms can be described using various media such as natural languages, flow diagrams, or perfectly standardized programming languages. The problem with using natural languages is that they can sometimes be ambiguous, while the problem of using programming languages is that they can be too restrictive depending on the type of behavior we want to describe.

Pseudocode is a midpoint:

- It is similar to a programming language but more informal.
- It uses a mix of natural language sentences, programming language instructions, and some keywords that define basic structures.

B.2 Operations and Control Structures

Two important aspects of programming languages and of pseudocode are the **actions** that can be performed and the supported **control structures**. Table B.1 depicts the actions and control structures of the particular pseudocode used in this book.

B.2.1 Assignments

An *assignment* is the action of assigning a value to a variable or a signal. In order to avoid confusion between the symbols that represent the assignment of variables and signals in VHDL (the hardware

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9

Table B.1 Actions and control structures

Actions	Control structures
(a) Assignments (b) Operations: Comparison Logic operations Arithmetic operations	(a) Selection (decision): If .. then .. else Case (b) Iteration (cycles): While For Loop (c) Functions and procedures

description language used in this course) they are represented by a simple equal sign (=). Other symbols (<=, :=) are frequently used in the literature and in certain programming languages.

Example B.1 The following set of sentences:

```
X = 3;
Y = 2;
Z = 1;
Y = 2·X + Y + Z;
```

assigns the value 3 to X and the value 1 to Z. Variable Y is set to 2 in the second sentence (Y = 2), but a new value 9 (2 · 3 + 2 + 1) is assigned to Y in the fourth sentence (Y = 2 · X + Y + Z).

B.2.2 Operations

The pseudocode must allow performing comparisons and logical and arithmetic operations. Tables B.2, B.3 and B.4 summarize the available operations.

B.2.2.1 Comparison Operators

The comparison operators are the classical ones: smaller than (<), greater than (>), equal to (=), smaller than or equal to (we will use indistinctly the symbols ≤ or <=), greater than or equal to (≥ or >=), and different from (≠ or /=). The same symbol (=) is used both in the comparison operation "equal to" and to assign values to variables or signals, but they are easily distinguishable by context.

B.2.2.2 Logical Operators

Logical (also called Boolean) operators have one (unary operator) or two (binary operator) operands and the result is a logical value TRUE or FALSE.

- The binary "**and**" operator has two operands, delivering the result TRUE if both operands are TRUE, and FALSE in any other case. For instance, the result of

  ("*x* is odd" and "*x* is less than or equal to 7")

  is TRUE when $x \in \{1,3,5,7\}$ and FALSE for any other value of *x*.
- Similarly, the binary "**or**" operator has two operands and delivers the result TRUE if at least one of the operands is TRUE, and FALSE when both operands are FALSE. In this case, the result of

  ("*x* is odd" or "*x* is less than or equal to 7")

  is TRUE when $x \in \{0,1,2,3,4,5,6,7,9,11,13, \ldots$ (any other odd number)$\}$ and FALSE when $x \in \{8,10,12,14, \ldots$ (any other even number)$\}$.

Table B.2 Comparison operations

Operator	Operation
$<$	Smaller than
$>$	Greater than
$=$	Equal to
$\leq$ or <=	Smaller than or equal to
$\geq$ or >=	Greater than or equal to
$\neq$ or /=	Different from

Table B.3 Logical operations

Operator	Operation
and	Logical product
or	Logical sum
not	Negation

Table B.4 Arithmetic operations

Operator	Operation
+	Sum
−	Difference
* or ·	Product
/	Division
**, ^ or superscript	Exponentiation

- The unary "**not**" operator has a single operand. When the operand is TRUE the result is FALSE, and when the operand is FALSE the result is TRUE. As an example, "x is odd" is equivalent to "not (x is even)".

B.2.2.3 Arithmetic Operators

- The classical arithmetic operators are sum (addition), difference (subtraction), product, and division. Sometimes a fifth operator, the exponentiation, is introduced. The operation "a to the power b" is represented indistinctly by a^b, $a**b$, or $a\text{^}b$. Roots are represented as numbers raised to a fractional power as, for instance, $a^{1/2} = \sqrt{a}$ or $a^{(1/4)} = \sqrt[4]{a}$.
- Arithmetic expressions are evaluated following the well-known operation precedence rules: exponentiation and roots precede multiplication and division which, in turns, precede addition and subtraction. Parentheses or brackets are used to explicitly denote precedences by grouping parts of an expression that should be evaluated first.

B.2.3 Control Structures

Control structures allow executing the pseudocode instructions in a different order depending on some conditions. There are three basic control structures: the selection structure, the iteration structure, and the functions and procedures.

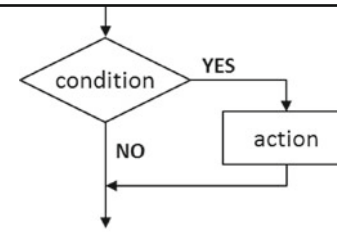
B.2.3.1 Selection Structures

There are two types of selection or branching structures: if . . . then . . . else and case:

If . . . **then** and **if** . . . **then** . . . **else**

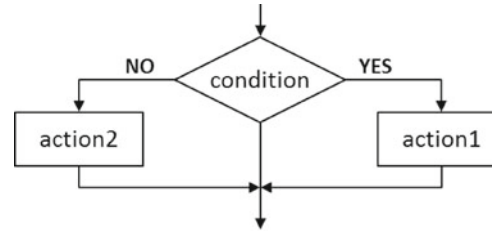
if *condition* **then** *action/s*;
end if;

If *condition* holds, *action* (or a set of *actions*) is performed. The corresponding data flow diagram is shown on the right.



if *condition* **then** *action/s1*;
else *action/s2*;
end if;

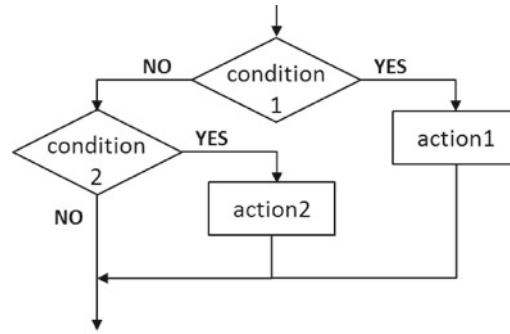
If *condition* holds, *action1* (or the set of *actions1*) is performed. If *condition* does not hold, *action/s2* is performed. The corresponding data flow diagram is shown on the right.



If . . . then . . . else structures can be nested as shown below:

if *condition1* **then** *action/s1*;
elsif *condition2* **then** *action/s2*;
end if;

if *condition1* holds, *action/s1* is/are performed. If *condition1* does not hold, then, if *condition2* holds, *actions/s2* is/are performed. The corresponding data flow diagram is shown below. Multiple nesting levels are allowed.

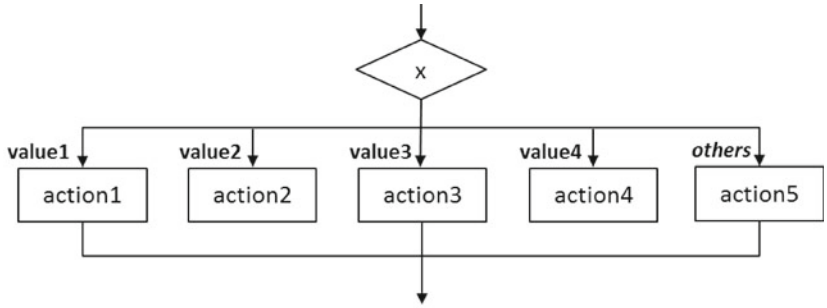


B.2.3.2 Case

The **case** statement is particularly useful when different actions must be performed depending on the value of a given variable or expression. For example, assume that variable *x* can take four values *value1*, *value2*, *value3*, and *value4*. Then, sequences of actions can be selected in function of the value of *x* by using the case statement:

```
case x is
        when "value1" => action1; -- see comment-1
        when "value2" => action2;
        when "value3" => action3;
        when "value4" => action4;
        when others   => action5; -- see comment-2
end case;
```

Symbol "--" indicates that what follows is a comment.
Comment-1: The clause "then" is often used instead of symbol "=>".
Comment-2: "Others" is optional. It means "when x takes any other value different from value1, 2, 3 or 4, then action5 is performed".



When the same action is associated with several different values of the branching condition (*expression* in the next example), the "case" instruction can be compacted in an informal and intuitive way; for example

```
case expression is
                when "value1" => action1;
                when "value2" or "value3" => action2;
                when "value4" to "value8" => action3;
                …
                when others   => action5;
end case;
```

Example B.2 (Nested if . . . then . . . else) Assume that you must compute *y* equal to the truncated value of *x*/2 so that the result is always an integer. If *x* is positive, *y* will be the greatest integer smaller than or equal to *x* divided by 2. If *x* is negative, then *y* is equal to the smallest integer greater than or equal to *x* divided by 2:

$$x > 0: \quad y = \lfloor x/2 \rfloor$$

$$x < 0: \quad y = \lceil x/2 \rceil$$

Thus, if

- x is even, then $y = x/2$;
- x is odd and positive, then $y = (x - 1)/2$;
- x is odd and negative, then $y = (x + 1)/2$.

For example, if $x = 10$ then $y = 10/2 = 5$ and if $x = -10$ then $y = -10/2 = -5$; if $x = 7$ then $y = (7 - 1)/2 = 3$; if $x = -7$ then $y = (-7 + 1)/2 = -3$.

Thus, the following algorithm computes the integer value of $x/2$:

```
if (x is even) then y = x/2;
     elsif (x is negative) then y = (x+1)/2;
     else  y = (x -1)/2;
end if;
```

Example B.3 (Case) Let us see a second example. Assume $x \in \{0,1,2,3,4,5,6,7,8,9\}$ and you want to generate the binary representation of $x : y = x_2$. The binary numbering system is explained in Appendix C. A straightforward solution is to consult Table C.1: it gives the equivalence between the representation of naturals 0–15 in base 10 and in base 2. The entries of Table C.1 that correspond to numbers 0–9 can be easily described by the following case statement:

```
case x is
     when 0 => y=0000;
     when 1 => y=0001;
     when 2 => y=0010;
     when 3 => y=0011;
     when 4 => y=0100;
     when 5 => y=0101;
     when 6 => y=0110;
     when 7 => y=0111;
     when 8 => y=1000;
     when 9 => y=1001;
end case;
```

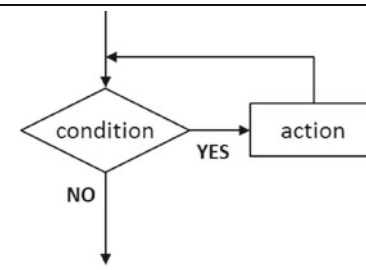
B.2.3.3 Iteration Structures

Another important type of control structure is the iteration. We will see two examples: the **while-loop** and the **for-loop**.

While . . . loop

while *condition* **loop**
action/s;
end loop;

A **while** structure performs an *action* (or a set of *actions*) as long as *condition* holds, as shown in the data flow diagram.

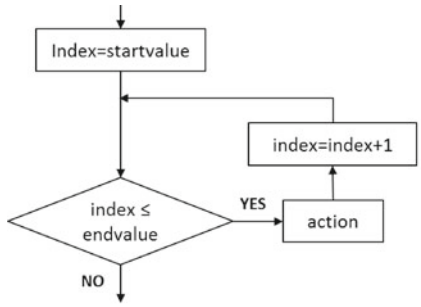


For . . . loop

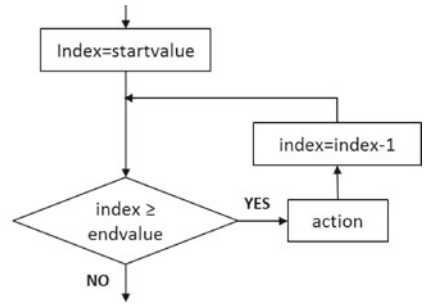
for *index* **in** *startvalue* **to** *endvalue* **loop**
action/s;
end loop;

Index is initially equal to *startvalue* and then increased or decreased by 1 depending on whether *startvalue* is greater or less than *endvalue*. While index is between *startvalue* and *endvalue*, *action/s* is/are performed and *index* is incremented or decremented by 1 as the following data flow diagrams show.

startvalue < endvalue



startvalue > endvalue



Example B.4 (While-Loop) Assume that we have previously defined two 8-component vectors:

$$A = a(0), a(1), \ldots, a(7),$$

$$X = x(0), x(1), \ldots, x(7),$$

and we want to compute the scalar product of A and X, that is,

$$y = a(0) \cdot x(0) + a(1) \cdot x(1) + \cdots + a(7) \cdot x(7)$$

A **while** structure can be used to describe the operation as follows:

```
i=0; acc=0;
   while i≤7 loop
      acc = acc + a(i)·x(i);
      i = i +1;
   end loop;
y = acc;
```

The first two statements initialize variables i and acc to 0. As the condition $i \leq 7$ holds ($i = 0$), the actions are performed; so, i is set to 1 ($i = 0 + 1$) and acc takes the value $a(0) \cdot x(0)$. Then the condition is checked again; $1 \leq 7$, so that the actions are performed again: $acc = a(0) \cdot x(0) + a(1) \cdot x(1)$ and $i = 2$. This process is repeated again and again until $i = 8$. When $i = 8$ the condition does not hold, so the execution of the loop ends and the last value of $acc = a(0) \cdot x(0) + a(1) \cdot x(1) + \ldots + a(7) \cdot x(7)$ is loaded into variable y.

Example B.5 (For-Loop) The same computation $y = a(0) \cdot x(0) + a(1) \cdot x(1) + \cdots + a(7) \cdot x(7)$ can be performed using a **for** ... **loop** structure:

```
acc=0;
for i in 0 to 7 loop
   acc = acc + a(i)·x(i);
end loop;
y = acc;
```

Index i is initially set to 0 and automatically incremented by 1 each time the loop is executed. As *acc* has been initialized to 0, the first time the loop is executed $i = 0$ and $acc = a(0) \cdot x(0)$. The last time the loop is executed is when $i = 7$ and thus $acc = a(0) \cdot x(0) + a(1) \cdot x(1) + \ldots + a(7) \cdot x(7)$. At this point the loop ends and the last value of *acc* is loaded into y.

B.2.3.4 Functions and Procedures

Functions and procedures, also called subroutines, are blocks of pseudocode performing a specific computation that can be used more than once in other parts of the pseudocode.

The difference between functions and procedures can be sometimes confusing. For example, in Pascal, a procedure is a function that does not return a value. In C, everything is a function, but if it does not return a value, it is a "void" function. We will use the most commonly accepted meaning for those terms. A function is a block of pseudocode that computes a mathematical function and returns a result. A procedure is a block of pseudocode that performs some task and returns the control to the main algorithm when finished. Let us see both more deeply.

B.2.3.5 Function

A function is formally defined as follows:

function *name* (*formal input parameters*)
    ...
    ...
    return (*expression* or *value*);
    ...
end function;

Within the main algorithm a function is called by writing its name and by giving actual values to the formal parameters:

...
x = *name* (*actual input parameters*);
...

The value computed within the function is assigned to variable x. The call can also be a part of an expression:

...
x = 2 · (*name* (*actual input parameters*))^2;
...

In this case the value computed within the function is squared and multiplied by 2 before being assigned to variable x.

Example B.6 (Function) Assume that we have defined the following function:

```
function test (x, y, data1, data2)
   if (x^2 - 2·y) < 0 then return (data1 - data2);
   else  return (data1 + data2);
   end if;
end function;
```

and we use the function in the following main pseudocode program:

```
...
a = test (in1, in2, 5, 4);
b = a · test (in3, in4, 0, 8);
c = a + b;
...
```

The result of these sentences when $in1 = 8$, $in2 = 4$, $in3 = 2$, and $in4 = 15$ will be $a = 9$ because $(in1^2 - 2 \cdot in2)$ is a positive number and 5+4 = 9, and $b = -72$ so that $(in3^2 - 2 \cdot in4)$ is less than 0 and $9.(0 - 8) = -72$.

B.2.3.6 Procedure

A procedure is defined by a *name* and by an optional set of *formal parameters*:

procedure *name* (*formal parameters*)
   …
   (**return**)
   …
end procedure;

When a procedure is called, the list of formal parameters is substituted by the actual parameters: variables used as operands within the procedure computations (input parameters) and variables whose values are updated with procedure computation results (output parameters). Every time the procedure finds a "return" or an "end procedure" sentence, the flow control goes back to the piece of pseudocode from which it was called, more precisely to the sentence below the call.

The procedure can be called from any piece of pseudocode simply by writing its name and the list of values with which computations will be done (actual parameters):

name (*actual parameters*)

Example B.7 (Procedure) Let us see an example: we want to compute the value of

$$z = a.\sqrt{x} + b.\sqrt{y}$$

with an accuracy of 16 fractional digits. An algorithm computing the square root of a number can be developed and encapsulated within a procedure "square_root" with parameters x, y, and n. Procedure square_root will compute the square root of x with n fractional digits and will return the result into y.

```
procedure square_root (x, y, n)
    ...
    ...
end procedure;
```

Now, this procedure can be used in the main algorithm to compute the square root of x and the square root of y without writing twice the sentences necessary to calculate the square root:

```
square_root (x,u,16);-- computes u ← √x with accuracy=16
u = u · a;                -- u ← a.√x
square_root (y,v,16); -- computes v ← √y with accuracy=16
v = b . v;                -- v ← b.√y
z = u + v;                -- z = a.√x + b.√y
...
```

The use of procedures not only facilitates the design of pseudocode but also improves the understanding avoiding the unnecessary repetition of sentences.

Appendix C: Binary Numeration System

Mercè Rullán

Computers and other electronic systems receive, store, process, and transmit data of different types: numbers, characters, sounds, pictures, etc. The common point is that all those data are encoded using 0s and 1s because computer technology relies on devices having two stable states which are associated with bits (binary digits) 0 and 1, respectively. In particular, numbers must also be represented by 0s and 1s, and the binary numeration system (base 2) offers the possibility not only to represent numbers but also to perform arithmetic operations using well-known and reliable algorithms.

This appendix is a short review of those fundamental concepts of the binary numeration system that all digital systems designer should know. Due to the convenience, sometimes, of expressing the numbers in a more compact form, a brief explanation of the hexadecimal system has been included.

C.1 Numeration Systems

Most commonly used numeration systems are positional: numbers are represented by strings (sequences) of digits and a weight is associated with every digit position. For example, in decimal (base 10 numeration system), 663 represents the following number:

$$663 = 6 \cdot 10^2 + 6 \cdot 10^1 + 3 \cdot 10^0$$

Thus, in the preceding expression, digit 6 has two different weights depending on its position: the leftmost 6, located in the position of the hundreds, has a weight equal to 10^2, while the 6 in the middle, located in the position of the tens, has a weight equal to 10^1.

The decimal system uses 10 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, and the n-digit sequence $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ represents the number

$$X = \sum_{i=0}^{n-1} x_i \cdot 10^i$$

More generally, the base b numeration system, being $b \geq 2$ a natural (non-negative integer), uses b digits $0, 1, 2, \ldots, b-1$. The weights associated with each digit position, starting with the rightmost digit, are $b^0, b^1, b^2, b^3, \ldots$ and so on. Thus, the base-b n-digit sequence $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ represents the number

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9

$$X = \sum_{i=0}^{n-1} x_i \cdot b^i \quad \text{where} \quad x_i \in \{0, 1, 2, \ldots, b-1\}$$

C.1.1 Binary Numeration System

The binary numeration system ($b = 2$) uses two binary digits 0 and 1. The universally known term "bit" is just a contraction of "BInary digiT".

Thus, the n-bit sequence $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ represents the number

$$X = \sum_{i=0}^{n-1} x_i \cdot 2^i, \quad \text{where} \quad x_i \in \{0, 1\}$$

For example, the binary number 1101 represents the decimal number

$$1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 13$$

In what follows the following notation will be used:

$$1101_2 = 13_{10}$$

C.1.1.1 Range

With n bits all natural numbers from 0 to $2^n - 1$ can be represented. Formally we will say that the range of representation of natural numbers in base 2 is the interval 0 to $2^n - 1$, where n is the number of bits.

Example C.1 For $n = 4$, the smallest natural number that can be represented is 0 (0000 in binary), and the greatest is $2^4 - 1 = 15$ (1111 in binary). Table C.1 shows the binary-decimal equivalence of these 16 numbers:

In the binary system, n bits can encode up to 2^n different values. As we have seen above, four bits allow defining $2^4 = 16$ different numbers, five bits allow to code $2^5 = 32$ numbers, and so on.

Now the inverse problem can be stated: How many bits are necessary to represent the decimal number 48? With five bits we can represent any natural number from 0 to $2^5 - 1 = 31$. With an additional bit ($n = 6$) we can represent any number up to $2^6 - 1 = 63$. As 48 is greater than 31 but less than 63, we can conclude that six bits are required to represent the number 48. Thus, the minimum number n of bits required to represent a natural number X is defined by the following relation:

$$2^{n-1} \leq X < 2^n$$

Table C.1 Binary-decimal equivalence with $n = 4$

Binary	Decimal	Binary	Decimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

C.1.2 Hexadecimal Numeration System

Sometimes the use of a base $2^4 = 16$ numeration system is very useful because it allows describing binary numbers in a compact form. It is the so-called hexadecimal or base-16 numeration system.

The hexadecimal system ($b = 16$) uses 16 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F, where letters A, B, C, D, E, and F represent the decimal values 10, 11, 12, 13, 14, and 15, respectively. Thus, the hexadecimal n-digit sequence $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ represents the number

$$X = \sum_{i=0}^{n-1} x_i \cdot 16^i, \quad x_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

For example, the hexadecimal number 3A9F, where A stands for 10 and F stands for 15, represents the decimal number:

$$15 \cdot 16^0 + 9 \cdot 16^1 + 10 \cdot 16^2 + 3 \cdot 16^3$$

Performing these calculations in base 10, we get that 3A9F in hexadecimal is equivalent to 15,007 in base-10:

$$3\text{A9F}_{16} = 15 \cdot 16^0 + 9 \cdot 16^1 + 10 \cdot 16^2 + 3 \cdot 16^3 = 15,007_{10}$$

C.1.2.1 Range

Following the same reasoning we used to define the range of representation of the binary system, with n hexadecimal digits all natural numbers from 0 to $16^n - 1$ can be represented. Formally we will say that the range of representation of natural numbers in base 16 is the interval 0 to $16^n - 1$, where n is the number of hexadecimal digits. As before, the number n of hexadecimal digits required in order to represent a natural number X is defined by the following relation:

$$16^{n-1} \leq X < 16^n$$

Table C.2 shows the equivalence of the 16 hexadecimal digits in decimal and in binary:

Table C.2 Decimal and binary equivalences of the hexadecimal digits

Hexadecimal	Decimal	Binary	Hexadecimal	Decimal	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

C.2 Base Conversion

Any number $X = x_{n-1}\, x_{n-2} \ldots x_1 x_0$ expressed in base $b1$ can be converted to base $b2$ by representing $b1$ and the n digits x_i in base $b2$, and by computing in base $b2$ the expression

$$\sum_{i=0}^{n-1} x_i \cdot b1^i (\text{computed in base } b2)$$

This universal method poses two problems. The first is the conversion of $b1$ and digits x_i to base $b2$. The second is that the execution of operations in base $b2$ can be somewhat cumbersome. Fortunately, if we are only interested in working with bases 10, 2, and 16, more friendly methods exist.

C.2.1 Conversion Between Binary and Hexadecimal Systems

As a matter of fact, the hexadecimal system is nothing else than an easier and more compact way to represent numbers in binary. The conversion from one system to the other is straightforward.

C.2.1.1 From Base 16 to Base 2

Consider a hexadecimal number $X = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$. Each hexadecimal digit x_i can be represented in base 2 with four bits as shown in Table C.2. To convert X to binary we just replace each hexadecimal digit by the equivalent binary 4-bit number.

Example C.2 Assume that we want to convert to binary the hexadecimal number 3A9. We simply substitute each hexadecimal digit by its 4-bit equivalent value (Table C.2):

Hexadecimal	3	A	9
Binary	0011	1010	1001

$$3A9_{16} = 001110101001_2$$

Example C.3 What is the binary representation of hexadecimal number CAFE?

Hexadecimal	C	A	F	E
Binary	1100	1010	1111	1110

$$CAFE_{16} = 1100101011111110_2$$

How many bits are required to represent an n-digit hexadecimal number? $4 \cdot n$.

C.2.1.2 From Base 2 to Base 16

Conversely, to translate a binary number to a hexadecimal one, we partition the binary number into 4-bit vectors, from right to left (from the least to the most significant bit), and we replace each group by its equivalent hexadecimal digit.

Example C.4 Convert 100101110100101 to hexadecimal.

First step: Separate the binary number in groups of 4 bits, starting from the right. If the leftmost group has less than four bits, just consider that the empty positions are 0s:

Binary	.100	1011	1010	0101

Second step: Substitute each group of four bits by the equivalent hexadecimal digit (Table C.2):

Hexadecimal	4	B	A	5
Binary	.100	1011	1010	0101

$$100101110100101_2 = 4BA5_{16}$$

C.2.2 Conversion Between Binary and Decimal Systems

C.2.2.1 From Base 2 to Base 10

Conversion from base 2 to base 10 has already been explained in Sect. C.1.1: just represent the binary number as a sum of bits x_i multiplied by 2^i where i is the position occupied by the bit x_i, and perform the calculations in base 10.

Example C.5

$$10011101_2 = \\ 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 = 157_{10}.$$

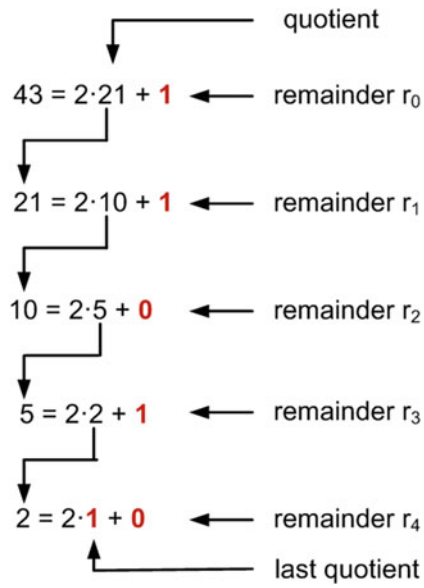
C.2.2.2 From Base 10 to Base 2

The conversion from base 10 to base 2 is not as simple as from base 2 to base 10. The conversion algorithm consists of a sequence of integer divisions by 2 with quotient and remainder. Let X be the number in base 10 that we want to convert to base 2. The method of successive divisions is the following[4]:

```
z = X; i = 0;
while z ≠ 1 loop
    q = ⌊z/2⌋;              -- q ← quotient
    r_i = z - 2·q;          -- r_i ← remainder
    z = q;
    i = i + 1;
end while;
X(base 2) = z r_i r_i-1... r_0   -- z = last quotient (= 1)
```

[4] Appendix B explains the basics of pseudocode.

Example C.6 Convert 43 to binary. The sequence of divisions is as follows:



Thus

$$43_{10} = 101011_2.$$

To check whether the result is correct we can perform the inverse operation and convert 101011 to base 10:

$$101011_2 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^3 + 1 \cdot 2^5 = 43_{10}$$

C.3 Binary Arithmetic: Addition and Subtraction in Base 2

It remains to be seen how addition and subtraction are performed in binary. Actually, the way to perform these operations is just an adaptation of the classical algorithm that we use to add and subtract numbers in base 10.

C.3.1 Addition

In base 2, when we add up two bits, there are four possibilities:

$$0 + 0 = 0,$$

$$0 + 1 = 1 + 0 = 1,$$

$$1 + 1 = \mathbf{10}\ (1 + 1 \text{ is } 2, \text{and } 2 \text{ is equal to } 10 \text{ in base } 2).$$

Assume that we want to add $A = \ 10100101$ and $B = \ 1010111$. We will follow the same steps that we use to perform a sum in base 10:

- We align the two addends and begin to add bits belonging to the same column, starting from the rightmost column.
- When the sum of the bits of a column is greater than 1, that is, 10 or 11, we say that the sum bit corresponding to this column is 0 or 1 and that a carry must be transferred (carried) to the next column.

Consider the rightmost column: $1 + 1 = 10$; so a 0 is written in the rightmost position of the result and a carry $= 1$ is generated.

```
          1     → carry
  1 0 1 0 0 1 0 1 = A
+ 1 0 1 0 1 1 1 = B
-----------------
          0
```

At next step we add 0 and 1 (second column values) plus the generated carry: $0 + 1 + 1 = 10$. We write 0 in the second result position and a new carry is generated.

```
        1 1     → carry
  1 0 1 0 0 1 0 1 = A
+ 1 0 1 0 1 1 1 = B
-----------------
        0 0
```

Now we have to compute $1 + 1 + 1 = 11$. We write 1 in the third result position and a new carry is generated.

```
      1 1 1     → carry
  1 0 1 0 0 1 0 1 = A
+ 1 0 1 0 1 1 1 = B
-----------------
      1 0 0
```

... and so on. From this point to the end no new carries are generated, and the result of $A + B$ is:

```
      1 1 1     → carry
  1 0 1 0 0 1 0 1 = A
+ 1 0 1 0 1 1 1 = B
-----------------
  1 1 1 1 1 1 0 0
```

More formally, assume that we want to compute $S = A + B$, where $A = a_{n-1}\ a_{n-2} \ldots a_1\ a_0$, $B = b_{n-1}\ b_{n-2} \ldots b_1\ b_0$, and $S = s_n\ s_{n-1}\ s_{n-2} \ldots s_1\ s_0$. The following algorithm can be used to perform the operation:

```
carry0=0;
for i in 0 to n-1 loop
   sum = ai + bi + carryi;
   if sum = 0 then si = 0; carryi+1 = 0;
   elsif sum = 1 then si = 1; carryi+1 = 0;
   elsif sum = 2 then si = 0; carryi+1 = 1;
   else  si = 1; carryi+1 = 0;
   end if;
end loop;
```

C.3.2 Subtraction

When computing the difference between two numbers there are also four possibilities:

$$0 - 0 = 0,$$

$$1 - 0 = 1,$$

$$1 - 1 = 0,$$

$$0 - 1 = \mathbf{11}.$$

In the fourth case, 0 minus 1 is equal to -1 and this value is expressed as

$$1 \cdot \left(-2^1\right) + 1 \cdot 2^0 = \ -2 + 1 = -1$$

that is given a negative weight to the leftmost bit. So, when subtracting 1 from 0, the result bit is 1 and a borrow must be transferred to (borrowed from) the next column. Assume that we want to compute $A - B$, where $A =$ 10100101 and $B =$ 1010111. We align the numbers and begin to subtract the bits of A and B,

$$\begin{array}{r} 1\,0\,1\,0\,0\,1\,0\,1 = \text{A} \\ -\ \ 1\,0\,1\,0\,1\,1\,1 = \text{B} \\ \hline \qquad\quad 1 \quad \rightarrow \text{borrow} \\ \qquad\quad 1\,0 \end{array}$$

from right to left. The rightmost column is easy to process: $1 - 1 = 0$. To the next column corresponds $0 - 1 = -1$, so we write 1 in the second result position and a borrow is generated. In the next step we compute $1 - 1 - 1 = -1$. Thus, we write 1 in the third result position and generate a new borrow.

$$\begin{array}{r} 1\,0\,1\,0\,0\,1\,0\,1 = \text{A} \\ -\ \ 1\,0\,1\,0\,1\,1\,1 = \text{B} \\ \hline \qquad\quad 1\,1 \quad \rightarrow \text{borrow} \\ \qquad\quad 1\,1\,0 \end{array}$$

At next step we compute $0 - 0 - 1 = -1$, so we write 1 in the fourth result position and a borrow is generated.

$$\begin{array}{r} 1\,0\,1\,0\,0\,1\,0\,1 = \text{A} \\ -\ \ 1\,0\,1\,0\,1\,1\,1 = \text{B} \\ \hline \qquad\quad 1\,1\,1 \quad \rightarrow \text{borrow} \\ \qquad\quad 1\,1\,1\,0 \end{array}$$

The next computation is $0 - 1 - 1 = -2$. This value is expressed as

$$1 \cdot \left(-2^1\right) + 0 \cdot 2^0 = -2 + 0 = -2.$$

So, when subtracting 2 from 0, the result bit is 0 and a borrow must be transferred to the next column.

```
  1 0 1 0 0 1 0 1 = A
- 1 0 1 0 1 1 1 = B
    1 1 1 1     → borrow
  ---------------
      0 1 1 1 0
```

. . . and so on. Finally we get:

```
  1 0 1 0 0 1 0 1 = A
- 1 0 1 0 1 1 1 = B
  1   1 1 1 1     → borrow
  ---------------
  0 1 0 0 1 1 1 0
```

More formally, assume that we want to compute $D = A - B$, where $A = a_{n-1}a_{n-2}\ldots a_1a_0$, $B = b_{n-1}b_{n-2}\ldots b_1b_0$, and $D = d_{n-1}d_{n-2}\ldots d_1d_0$. The following algorithm can be used to perform the operation:

```
borrow0=0;
for i in 0 to n-1 loop
   dif = ai - bi - borrowi;
   if dif = 0 then di = 0; borrowi+1 = 0;
   elsif dif = 1 then di = 1; borrowi+1 = 0;
   elsif dif = -1 then di = 1; borrowi+1 = 1;
   else  di = 0; borrowi+1 = 1;
   end if;
end loop;
```

A final comment: In the explanation of the subtraction operation we have assumed that the minuend is greater than or equal to the subtrahend ($A \geq B$), so that the difference $A - B$ is a nonnegative number D. In this appendix, only natural numbers (non-negative integers) have been considered. In order to represent negative numbers, other methods beyond the scope of this appendix, such as the sign and magnitude or the 2's complement representation, are used. Anyway, it is worthwhile to observe that if the preceding algorithm is executed with operands A and B such that $A < B$, then $borrow_n = 1$ and the difference $A - B$ is equal to $-2^n + D$ (actually the 2's complement representation of $A - B$).

Index

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9